AFIT/GCE/ENG/93D-9

Interactive Control of a Parallel Simulation from a Remote Graphics Workstation

THESIS

Douglas Clifford Looney
Captain, United States Air Force

AFIT/GCE/ENG/93D-9

AFIT/GCE/ENG/93D-9

Interactive Control of a Parallel Simulation from a Remote Graphics Workstation

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Douglas Clifford Looney, B.S.E.E.

Captain, United States Air Force

14 December, 1993

Acknowledgements

Dr. Hartrum, Maj Sonnier, and LtCol Stytz are to be commended for their infinite patience and encouragement of this effort. A special debt of gratitude is owed to my thesis advisor, Dr Hartrum. His unceasing efforts to keep me running at 100 percent efficiency are greatly appreciated. Also included in the list of staff to whom I am in debt: Rick Norris, Dave Doak, Dan Zambon, and Bruce Clay.

Among the many students I have had the great pleasure of knowing while here, I must especially acknowledge the assistance of Flt. Lt. Paul Chase, "Uncle Larry" - Capt. Laurence Merkle, Capt Vince Droddy (who is a Latex demigod), Capt Wally Trachsel (who made BATTLESIM parallel), and Capt Seth Guanu.

Douglas Clifford Looney

Table of Contents

List of Figures

List of Tables

Abstract

Modern military commanders are faced with an overwhelming amount of intelligence data concerning the disposition of engaging forces. The sheer volume of data produced for a single planning scenario is an obstacle to the user as well as to the computer.

Today's commander requires an interface that displays a real-time, three-dimensional representation of the battlefield in order to assimilate the data for the management of a conflict. Parallel computation is required to complete the processing of this information in a timely manner. A network protocol is required to link the interface with the parallel simulation.

The purpose of this study is to improve user interaction through graphical representation of a parallel simulation. Each portion of the system, the user interface, the parallel simulation, and the network are discussed in this research. The concentration of the research deals with the parallel simulation's ability to accept and act upon user input without corrupting the integrity of its execution. This includes the ability to receive and act upon a *rewind* or *rollback* command from the user. Effectively, this command halts the parallel execution and rewinds is back for restart at a previous state and time.

Solutions to limitations in a conservative parallel simulation are developed and presented. Specifically, the ability of the parallel simulation to periodically save its state, and synchronize its separate logical processes for user input. All of the requirements for the user's control commands and the effect of these requirements upon the entire system are developed and demonstrated. This research provides an increased capability to assist in battlefield management and critical decision making processes.

Interactive Control of a Parallel Simulation from a Remote Graphics Workstation

I. Introduction

Development of large, complex computer simulations is providing both civilian and military organizations with a cheaper alternative to other testing implementations. The military can replace costly training exercises with battlefield simulations used to improve a unit's experience without deployment in the field. Design alternatives can be thoroughly analyzed in software well in advance of proceeding into costly production. Simulation has the advantage of shorter fix turnaround times and no costs associated with redesign of manufacturing or testing equipment.

Battlefield simulations can now be distributed over the globe, and can involve multi-service units from multiple countries. Scenarios can be planned for battlefield exercises and fine tuned prior to deployment in the field. This advance refinement saves money associated with training time, overhead for the test range, and safety.

The science and art of building these complex simulations is continually being investigated. Prior to development of a simulation, interaction requirements must be established. Many simulations require real time input during the execution itself. After having already developed a system, engineers might decide that real time input is an interaction requirement that needs to be added to the existing simulation.

This research is concerned with the improvement of existing and future battlefield simulations. Attention is focused specifically on the computationally intense planning scenarios. Theses scenarios are required to run on a parallel computer system in order to reduce execution time. This is a vital area of research. The ability to easily analyze different battle scenario alternatives within minutes can provide commanders with a strategic advantage in the field. This sort of planning would likely

take place in a distributed computing environment. Commanders would be accessing a computer resource some distance from their position.

The purpose of this research is to investigate and demonstrate the feasibility of real time interactive control of a parallel simulation in a distributed computing environment.

1.1 Background

1.1.1 User interaction. The user interaction requirements for battlefield simulations could vary widely, but can be loosely organized into three categories.

- The user may view the battle taking place from various viewpoints about the battlefield.
- The user may want to control the execution of the simulation while watching the display. This might involve stopping the simulation, replaying portions of it, or even modifying the objects in the simulation.
- The user may wish to participate in the simulation.

An interactive graphics tool is needed to provide such interaction with the battlefield simulation. Such a tool provides users with a three dimensional picture of the battlefield environment and the necessary input devices to control or participate in the simulation.

1.1.2 Event Driven Parallel Simulation. A large simulation may need to be parallelized in order to reduce its execution time. This requires multiple physical processors connected in some physical topology that allows them to communicate with each other for data, status, and control information. The parallelization of a simulation breaks it up into separate computational tasks referred to as logical processes (LPs). Each physical processor (referred to as a "node") may execute one or more logical processes.

The simulations of interest are event driven. The execution progresses as events are removed from a time ordered list called the *next event queue.* The simulation time itself is advanced to the next event time as it is removed from the queue. As an event is executed, the simulation updates

the simulation time to the time associated with the event. This event may generate future events which are inserted into the next event queue.

In a parallel simulation, each LP maintains its own simulation clock and next event queue. In order for the simulation to generate correct results, synchronization of these individual clocks is needed. Time synchronization takes place between the logical processes in the simulation through different approaches. These synchronization protocols are described by how aggressively they initiate message passing. An *optimistic* protocol initiates the passing as soon as it can. If it detects that the simulation clocks are out of synch, it initiates a recovery routine to return to correct synchronization. A *conservative* protocol will wait to pass a message until status information indicates that the execution will not be effected negatively by passing the message.

1.1.3 Network Communication. In order to take advantage of both specialized graphics and parallel processing hardware, a distributed system may be required. High performance parallel processing computers do not support high performance graphics. Some graphics workstations support a parallel processing capability, but their ability to scale up the number of processors is limited. The general solution used to get the best of both worlds involves taking advantage of network communication between these two.

There are many standards defining network communications. These standards dictate a particular format to packets of data to be transmitted between computers. Recent battlefield simulation research involves many distributed simulators executing together across a large network. A new standard called Distributed Interactive Simulation (DIS) is being drafted and it should allow many different types of machines to communicate and interact in widely diffused simulations, (see Section 2.6).

1.2 Problem Statement

A successful battlefield simulation requires an *interactive graphics tool* that allows the user to control a remote *parallel simulation* across a *local area network*. The distributed nature of the system can result in the actual simulation being at a different point in simulation time than what is being displayed on the graphics workstation. The problem of synchronizing the parallel simulation clocks further complicates the ability to interactively control the simulation. These problems must be solved if interactive control of a parallel simulation is to become a reality.

1.3 Research Objectives

The product of this research is an object oriented package that can be applied to any event driven parallel simulation for response to real time control commands from the user. The interactive simulation interface tool will initiate these control messages independent of the simulation.

Several real time capabilities were implemented for user control. The first capability consists of stopping the simulation and returning to a previously saved state of execution. Other control capabilities such as pause, continue, and restart of the simulation were also implemented. An ability to alter the simulation state by moving, adding or deleting objects has been addressed but not implemented.

1.4 Thesis Outline

Chapter Two reviews the system components in more detail, as well as summaries of technical literature relevant to this research. Chapter Three analyzes the requirements for the research. Chapter Four presents the general design of the project. Implementation of the design is described in Chapter Five. Chapter Six finalizes with the results, conclusions and performance data from the research. The appendices contain the users manual information for the interactive graphics tool and the parallel simulation.

II. Summary of Current Knowledge

2.1 Introduction

The purpose of this chapter is to review prior and ongoing studies relevant to this research. Prior research described in the following sections had resulted in an interactive graphics tool (Section 2.2) capable of displaying simulation data and interactively generating simulation control messages; and an object oriented parallelized battlefield simulation (Section 2.3) running in a non-interactive (batch) mode.

Section 2.5 discusses an alternative approach to time synchronization of parallel simulation, and Section 2.6 presents research efforts on a distributed simulation communication standard.

2.2 VISIT

In 1990 DeRouchey designed and developed the Visual Interactive Simulation Interface Tool, VISIT (7). VISIT was designed to provide the graphical representation and user interface for a remote simulation running on another processor. The system provides support for viewer interactions through a collection of commands allowing the user to generate messages to initialize, start, stop, abort, and restart a remote simulation. Other messages are generated for the viewer to have the ability to establish checkpoints. Upon reaching a checkpoint the viewer can step through the output display one unit at a time. Messages are also generated to create, destroy, or modify objects within the simulation.

VISIT runs on a Silicon Graphics workstation and accepts asynchronous time-stamped events using a local communications package. Based on the object information in each event, it extrapolates the position of moving objects while waiting for their next events. This extrapolation function is also known as dead reckoning. Its purpose is to provide smooth motion between updates in the position of the simulation's moving entities. Extrapolation is performed by predicting an object's

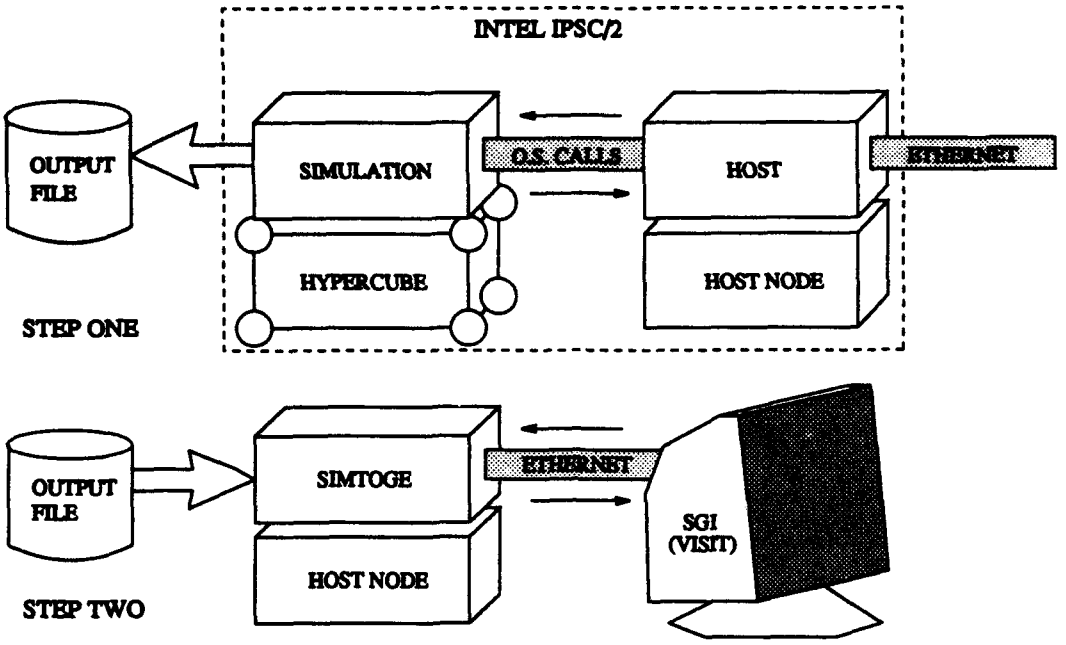

Figure 1. The former implementation

movement based on current time, position, orientation, and velocity vectors. VISIT does not implement acceleration vectors.

There are shortfalls to this method. First, the user may see a small jump in movement if the time of an update event does not match the increments used by the display system. Second, VISIT assumes events will arrive before their simulation time is reached by the graphics processor. If a "turn" event arrives after the extrapolation function has displayed the moving object past the event time, the user may see the object leap to its new position some distance past where it should have turned.

The simulation output is displayed in three dimensional graphics. VISIT allows the viewer to specify any set of viewing parameters. The viewer is also able to specify a terrain or playing surface and icons to represent the simulation objects. All displayable objects (icons, terrain, etc.) use the AFIT polygon file format (see appendix F). Distributed simulations require a common set of terrain data be used by all parties taking part in the simulation.

VISIT receives the display data messages from the network. In order to test and demonstrate VISIT, messages can be generated by a concurrently executing software tool running on the simulation processor. The tool is called SIMTOGE (Simulation to graphic engine). This tool reads a previously created simulation output file and sends the messages to the graphics workstation using a local protocol as shown in Figure 1.

2.3 BATTLESIM

BATTLESIM is an object oriented simulation written in C capable of generating data for VISIT (18). It runs in parallel on an Intel iPSC/2 Hypercube (20). It is an event driven simulation, developed independently from the interactive graphics system, that runs in batch mode.

The configuration of the iPSC/2 Hypercube is shown in Figure 1. A system manager processor controls the parallel cube nodes. This processor is referred to as the "host processor", and has the only access to the ethernet. It must act as an interface between the simulation running on the parallel processors and the control commands sent from the graphics workstation over the network.

The parallelization of the BATTLESIM application partitioned the computational tasks into logical processes (LPs). Each physical processor, or "node", can support from one to several logical processes. Each logical process keeps its own local next event queue and simulation clock. Events are retrieved and executed from the queue is time order. The logical processes communicate with each other through interface software that makes operating system calls between LPs. The logical processes have no knowledge as to the physical location of other logical processes, whether they are on their local node or another node, as shown in Figure 2.

BATTLESIM is designed to simulate objects (missiles, aircraft, and land vehicles) moving along predetermined route points until they either sense another object or a battlefield sector boundary. Objects may react to each other by attacking, evading, or continuing along their planned

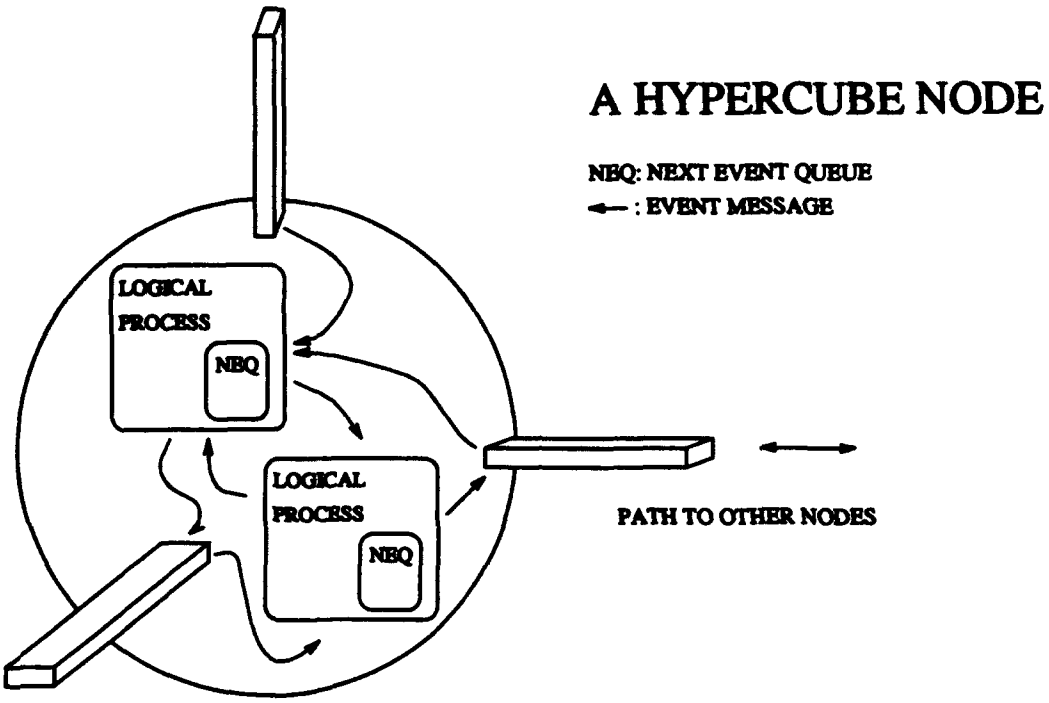

Figure 2. A Hypercube node

path of movement. Objects react to sector boundaries by determining the type of boundary-crossing event involved and then executing that event.

Scenario input files specify these objects and their attributes for each run of BATTLESIM. The files are retrieved at run time along with a mapping file that specifies the partitioning of the BATTLESIM application into separate logical processes (LPs). The specification of computational responsibility for each LP is done by dividing the geography into sectors and creating a mapping between LPs and sectors.

BATTLESIM is organized in a layered configuration within each logical process. The assigned sectors dictate any computation to be done by the owning logical process. Each sector has boundaries and neighboring sectors. Some information overlaps between these sectors, and is redundantly processed. Each sector also contains a set of objects such as vehicles, radar stations, or anything else to which the object abstraction can be applied. Under the sector layer is the object itself. The

object contains some common data and some data specific to its type. There exists further low level data consisting of route information and other data governing the mission of the object.

Synchronization takes place between the logical processes in the simulation using a conservative approach. This approach dictates that the separate logical processes execute events from time t only if they have received events from all possible inputs, and each received event has been time stamped at time t or greater.

BATTLESIM uses the conservative time synchronization protocol developed by Chandy and Misra (4, 5). This synchronization allows the simulation processing to proceed in correct order while distributed across the separate LPs.

2.4 SPECTRUM Simulation Testbed

BATTLESIM runs over the SPECTRUM (Simulation Protocol Evaluation on a Concurrent Testbed with Reusable Modules) testbed, designed to support the empirical comparative study of parallel simulation protocols and applications under a controlled environment (15)(16)(17). It provides an interface structure for each of the components of the simulation. The application, parallel machine, and the implementation of the simulation protocol interact through the process manager, the node manager, and the protocol filter(s) respectively. Filters can be altered or swapped relatively quickly to implement new simulation protocols. The object of the testbed is to provide this flexibility and baselined environment for empirical comparison studies of different applications interacting with different simulation protocols.

The testbed software used to run the application can provide the implementation of any synchronization protocol. It is the only piece of the simulation that requires modification to provide either conservative or optimistic protocols. In theory, the application itself would not have to suffer any modification to achieve a switch in protocol.

2.5 TIME WARP

An alternative method of time synchronization used in parallel discrete event simulation (PDES) is called Time Warp (13). It is an optimistic implementation related to virtual time, the local simulation time for each logical process in parallel simulation. The virtual time at any one process is not guaranteed nor expected to be the same for any two processes running in the application. Jefferson's implementation of time warp in a PDES has each process advance its clock on the receiving time of messages generated by events. His definition of an event consists of all actions that take place at process x and time t. Messages contain four pieces of information: the sender, virtual send time, receiver, and virtual receive time. Two rules apply to virtual time systems:

- The virtual send time of each message must be less than its virtual receive time.
- The virtual time of each event in a process must be less than the virtual time of the next event at that process.

Obeying these rules, event(x,t) may involve zero or more of the following operations:

- x may receive any number of messages stamped with receiver x and virtual receive time t, and read their contents;
- It may read its virtual clock;
- It may update its state variables;
- It may send any number of messages, all of which will automatically be stamped with sender x and virtual send time t.

The correct time warp implementation of virtual time dictates that messages are handled in the order of their virtual receive time, or time stamp. Because messages are not guaranteed to arrive in time stamp order, the time warp mechanism relies on a detect and recover method to insure correct processing of the simulation.

A process that receives a message with a time stamp that is less than its local virtual time attempts to roll-back in time. The roll-back process consists of two correction actions. The process first rolls back its virtual time (local simulation clock) and begins processing all over again. It also sends an anti-message for each message sent after the time stamp of the late message. The anti-message either cancels out the previous message waiting in the input queue of the next process, or causes the next process to initiate its own rollback procedure.

The time warp mechanism also employs a concept called Global Virtual Time (GVT). This is defined as the minimum of all the process' local virtual times at time t and virtual send times of all messages that have been sent but have not yet been processed at time t. Roll-back is limited to the GVT and it serves as the floor for the virtual times. It is a measure of system progress, and will never decrease.

2.6 Distributed Interactive Simulation

In February 1992, a draft for a new simulation network protocol was released by the Defense Advanced Research Projects Agency, DARPA (14). This new protocol, known as Distributed Interactive Simulation (DIS), was designed primarily to support DoD training by allowing computer simulators spread out over a large geographical area to interact in a team environment. The benefits of such a system include low training cost, joint service interaction, and safety. In order for DIS to succeed, operational guidelines and standards for inter-operation between the involved simulators have to be generated, potentially making it necessary to redesign some simulators to meet the new guidelines.

The DIS standard is the result of working groups established in different areas of distributed simulations. The standard has been submitted to IEEE and following approval will be submitted to the appropriate international agencies. DIS encapsulates the following properties:

- There is no need of a central computer for scheduling events or conflict resolution

- It consists of a distributed simulation environment using local copies of common terrain and model data bases.
- Entities maintain their own world view based on their own simulation.
- Entities are responsible for determining what is perceived.
- Entities broadcast data packets that demonstrate changes in their state.
- Entities employ dead reckoning to reduce communication processing.
- Entities closely correspond to weapon systems that they are modeling.

2.7 JMASS

The purpose of the Joint Modeling and Simulation System program is to provide the Department of Defense with a standardized digital simulation and modeling capability with which to develop, test, and assess the capabilities of weapon systems in their operational environment (1). The key to this capability is the development of a modeling and simulation architecture that is capable of supporting integration of digital models. Models of the weapon systems produced by the weapon system developer, threat systems developed by the intelligence community, and environmental effects developed by the scientific community would combine into a detailed simulation of weapon systems against threat systems in a simulated operational environment.

JMASS has provided a system/segment specification for the standardization of their digital simulation and modeling. The Execute Simulation Mode requirements are addressed specifically by this research. The control desired by the execute simulation mode specification includes commands to start, stop and save, restart, abort, pause and resume, and quit the simulation.

III. Requirements Analysis

3.1 Introduction

The requirements for this research are generated by the parallel simulation research faculty and staff at the Air Force Institute of Technology. This chapter describes the required functionality for an interactive parallel simulation system research tool. There are two major interacting components in the system, the graphics workstation with its display software, and the simulation computer with its simulation software.

During the running of the simulation, messages are created by the simulation computer and sent to the graphics workstation for display. The user can chose from two different modes for interaction: interaction with the remote simulation itself (remote mode) or interaction only with the local display (local mode). Remote mode commands are a super-set of local mode commands; they have the same capabilities plus additional commands to exercise simulation control.

Using simulation control commands in remote mode, user messages are issued from the graphics workstation to the simulation computer to affect the state of the simulation. The intent for remote mode commands is to allow user control and modification of the simulation. The user is able to perform interactions such as stopping of the simulation, entity manipulation, restart, pause, and continue.

Local mode commands control only the display of the simulation taking place. Interaction is limited to replay of simulation data that is retained in the workstation buffer. The intent for this mode is to allow the user's simulation to interact with other distributed simulations. Viewing parameter changes may be issued to the display software, but no messages are sent across the network to the simulation computer.

3.2 Interactive Graphics Requirements

This section describes the required functionality of the graphical interactive tool used to view and/or control a simulation being run on a remote processor.

3.2.1 Local Mode. In this mode the user is able to view the simulation taking place. Entities from the simulation are viewed on the workstation. The user is allowed to change only viewing parameters. The need for this mode exists since other distributed simulations may be affected by non-standard control messages placed on the network.

3.2.2 Remote mode. This mode allows the user to both view and control the simulation running on a separate processor. Simulation control commands consist of the following commands selected through the interactive display program. Each command results in the sending of a network message from the interactive graphics workstation to the simulation processor.

Pause. The command will pause the simulation, but allow display of the simulation time at which the user selected the pause command. This will allow the system to enter a state during which the user can observe the display and alter viewing parameters without the simulation continuing execution.

Continue. The simulation will continue execution from the simulation time at which it had been previously paused.

Stop. The simulation will stop and the display will react as in the pause command. The user will invoke this command before entering picking mode or selecting the restart command.

Picking Mode. After selecting the stop command, the user shall be able to select an entity and alter the entity's position and other data through the interactive display tool. The user will be able to enter this mode, select an entity, alter the entity's positional and aspect data, send the new data back to the simulation for update, and resume execution of the simulation with

the entity's new orientation. The entity will not be repositioned outside of its operational space. For example, the user will not be able to fly a submarine or submerge an aircraft.

Restart. After entering the stop command, the user can request that the simulation return to an earlier simulation time. The user will select the desired simulation time using the display interface, and that time will be returned to the simulation processor. The simulation must then update its state to the previous time selected and begin execution from that point.

Abort. The simulation processor will abort the simulation, and the interactive display program will cease upon receiving the ABORT message from the simulation.

3.2.3 Common Requirements. These requirements exist for both the remote mode and the local mode of the interactive graphics tool.The user will be presented with a display of the simulation and have the capability to manipulate the following qualities of the display.

Center of Interest. The user will be able to direct the view in any of the three dimensions.

Position. The user will be able pan the origin of viewing position in any of the three dimensions.

Field of View. The user will be able to increase and decrease the field of view.

Scale Factor. The user will be able to increase and decrease the size of the entity's icon representations.

Terrain Display. The user will be able to toggle the display of the terrain on and off.

Display Trails. The user will be able to toggle the display of entity path trails on and off.

3.3 Simulation Requirements

In order for the interactive control commands to be effective, the simulation must have the capabilities to respond accordingly. This research examines the extent of these capabilities and measures the amount of success that can be achieved with a parallel distributed simulation controlled

interactively from a remote graphics workstation. The degree of independence of the simulation from the controlling interactive tool should be maximized.

3.3.1 Local mode. The simulation processor broadcasts updates of the state of the simulation through network protocol messages. These messages are compatible with the network protocol format used by the remote graphics workstation. The messages contain the information necessary to display the state of the simulation on the graphics workstation.

3.3.2 Remote Mode. The simulation processor has the capability to receive control messages from the graphics workstation and make them available for access by the simulation processes. The simulation also has the ability to respond to the interactive commands outlined in Section 3.2.2. Response to any simulation control message results in a simulation time equal to the time displayed on the user's graphics workstation when the user invoked the control command. The state of the simulation is altered only if the user invokes the stop command followed by a restart command. While the simulation is stopped the user may select picking mode and manipulate simulation entities.

3.3.2.1 Simulation Control Commands. The simulation control commands are received by the simulation processor over the network. Each command is stamped with the simulation time at which the user selected the command. The following lists simulation control commands and their required responses:

Pause. The simulation processor receives the pause command message over the network with the simulation time stamp at which the command was selected. This command causes the simulation to execute a synchronized pause and wait for the continue command. This action keeps any new data from being transmitted across the network. The object of the command is not to require the simulation to change the state of its previous execution, only to temporarily

enter a holding state. The simulation will resume execution upon the receipt of the continue command.

Continue. The simulation will continue execution at the simulation time at which it had been previously paused.

Stop. The simulation processor receives the stop command message over the network. The simulation time at which the command was selected is stamped on the message. This command will cause the simulation to execute a synchronized stop and wait for the restart command. The user will invoke this command before entering picking mode or selecting the restart command.

Picking Mode. The simulation processor will receive the updated entity's state over the network, and pass the information to the simulation processes. The simulation will update itself and await the restart command to begin execution with this new state.

Restart. The simulation will return to a previous state. The user will select the desired simulation time using the display interface, and that time will be returned to the simulation processor. The simulation must then update its state to the previous time selected and be prepared to begin execution from that point. The simulation will begin the execution of the simulation with the new time plus new entity data (if picking mode was selected and an entity manipulated).

Abort. The simulation processor will gracefully end the simulation.

3.3.2.2 Time Synchronization Protocol. The time synchronization protocol remains independent of the implementation used in simulation control. The simulation control commands do not rely on the simulation's use of either an optimistic or conservative time synchronization protocol.

3.4 Distributed Interactive Simulation Compatibility

The simulation itself is required to be a full DIS compliant simulation, capable of both receiving and sending DIS compatible messages. In order to be fully compliant the simulation is required to perform dead reckoning for remote entities, as well as other battlefield interactions.

However, the control of the simulation through the sending of network packets requires the creation of unique packet formats. This requirement forces an implementation of a network protocol command packet that either does not conform to the DIS standard or is a modification to a specified DIS packet. This new packet is used to carry command messages to the simulation processor.

IV. Design

4.1 Design Goals

The overall research goal was to design and demonstrate a modular extension to a parallel simulation. This extension can be "easily" implemented to achieve interactive control. The requirements from Chapter 3 serve as a non-specific guide for this design. The demonstration of this design involves three different components. The BATTLESIM simulation, the VISIT simulation interface tool, and the Visithost program.

The foremost design task completed was to add a state saving capability to the BATTLESIM simulation. VISIT and Visithost were redesigned to initiate and pass the user commands in a real time environment. See Figure 3.

4.1.1 VISIT Interface Environment. The present design for VISIT provided almost all of its required functionality. Modification was needed to properly send restart time information to the simulation, but it otherwise met all user interaction requirements.

4.1.2 Visithost. The design of the previous Visithost program only communicated with the Hypercube. It was used to properly initiate and complete a parallel simulation from the host node of the IPSC/2. The redesign added the capabilities needed to communicate in the direction

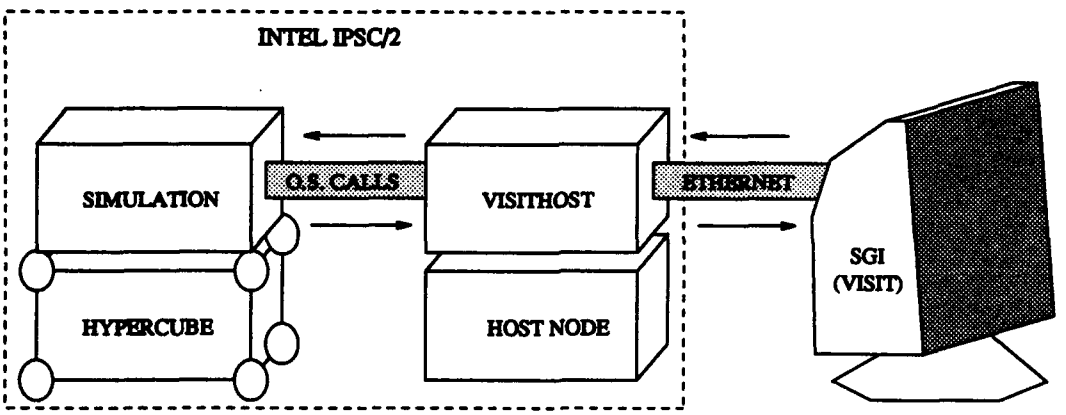

Figure 3. System Overview

of the networks. These capabilities are encapsulated in a single file. Thus, future upgrades to other network protocols will effect the routines in this file and not the Visithost program itself.

4.1.3 Application Capabilities. The design goals for extension of the BATTLESIM simulation are based on two concepts, synchronization of the logical processes, and the ability of the simulation to save its state. The state saving must be accomplished during a synchronized pause in the simulation. Other capabilities of stopping, restarting, and manipulating objects in the simulation extend from this ability to periodically save the state of the simulation and thus have a baseline from which to continue execution.

In the object oriented design of the BATTLESIM simulation, each layer of the object hierarchy needed to have four methods added to it to achieve the required save state functionality. In addition, the next event queue for each logical process required these same methods, along with methods to flush messages held in low level communication buffers upward into the next event queue. The flushing of these buffers was done after all of the logical processes had synchronized and ceased inter-process communication, but before the state of the next event queue had been recorded.

Record object state. When this method is called, it allocates enough contiguous memory space and makes a copy of the present state of the calling object. It calls any other lower level *record* methods. Each lower level record method returns to the calling record method with a reference pointer to its saved copy in contiguous memory. The record function must operate in this hierarchical fashion because the object oriented design paradigm dictates that only an object has knowledge of what it contains. For example, only the sector object will have access to the data needed to be saved for the vehicle objects that are inside that sector.

Restore object state. This method receives the previously recorded reference pointer from the calling routine and updates itself with the old state information. It calls lower level *restore* methods and *frees* the memory that had been holding the present state information for the simulation.

Free object state. The purpose for this method is to keep the simulation from running out of memory resources. When the simulation is given the command to restart with a simulation time that is less than the current time, it must free the memory associated with the current state. The previously saved states are held in a linked list. The list elements are identified by the simulation time at which they were recorded. The list elements are examined in reverse simulation time order and each one having a time stamp that is greater than the requested restart time is *freed*. The first list element having a time stamp that is less than the requested restart time is restored and the simulation is restarted from that point. The reason the *record* method specifies an allocation of contiguous memory space is to make it easy to deallocate it later. All that is needed is the pointer to the beginning of the space. No knowledge of the structure stored there is required.

Show object state. This method was used for testing purposes. It simply outputs the state of the object to a file for later examination.

4.1.4 Object Manipulation. The current object oriented design of the BATTLESIM simulation does not yet support full entity manipulation. In order for an entity to be altered, all reference to that entity must be updated across the simulation. The functionality for this task is not currently available or defined. Because an entity may migrate between logical processes during the simulation, the references to an entity in memory may be altered or duplicated in another location. The simulation does not track or manage these changes. Thus, an attempt to modify an entity's data may only effect one copy of the data. Knowledge of the where the entity is being tracked is required for manipulation.

In order to implement a design for user manipulation of migrating objects the simulation will have to encapsulate all the knowledge of the object inside the object. This includes the object's attributes, orientation, position, sensors and sensor data, and the route and mission of the object.

Message formats for the transfer of data will need to be designed in order to pass all information to the user at the remote workstation, and to receive updated copies in return.

The user's interface tool will require the capability to modify these elements and to check and maintain consistency between altered object data. The user should be prevented from moving an object's position outside of the object's operational space. The user should also not be allowed to co-locate two objects at the same position. The reality of the simulation must be maintained.

4.1.5 Network Interface. It remains a goal to update this design to the DIS network protocol standard. However, the research environment does not yet support DIS network communication. Although Section 3.4 states that interaction with entities generated by remote distributed simulations requires compliance with the DIS standard, the BATTLESIM simulation presently can not perform the required DIS object updates nor the network communication without environment support. Thus, the research undertaken by this thesis is not required to be fully DIS compliant. The network communication will continue using the TCP/IP protocols and object updates will not be modified to satisfy the DIS requirements.

4.2 System Design

4.2.1 VISIT. VISIT has a simple design that receives messages over an network, decodes the message, updates the state information held in a buffer for each object in the simulation, and finally updates the display of that information. Input from the user is executed if found in a polling loop. (See Figure 4).

4.2.2 Visithost and Battoge. Visithost is the governing routine that allows the user to select an application and run it on the Hypercube. After receiving all the needed initialization data, it loads the Hypercube and starts its execution. Visithost enters a loop at this point waiting for any input from the simulation by way of operating systems calls. Calls made by the simulation

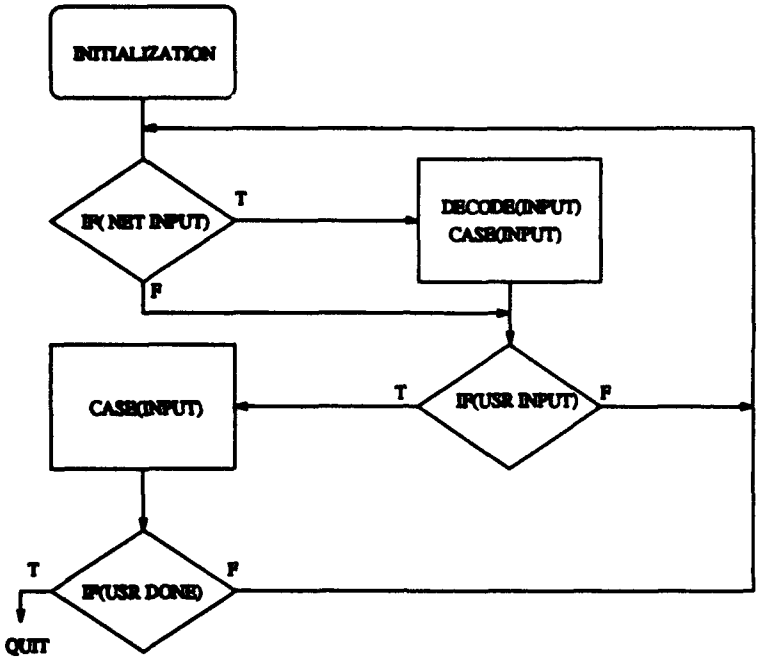

Figure 4. VISIT Flow Chart

that contain update messages are passed on to Battoge (BATTLESIM to graphics engine) routines where they are formatted and sent across the network to the graphics workstation for display as shown in Figure 5).

Once during each execution of the Visithost loop, another Battoge routine is called that checks the network for user input messages. These messages are packed into a buffer and sent to the simulation via the operating system calls. This allows communication between the user and the simulation. The need for such a handshake exists because the Hypercube itself does not have a direct connection to the network, (see Figure 3). Without this restriction the simulation could call the Battoge routines itself.

4.2.3 Object Migration. A complication to the design arises when an object crosses a sector boundary. This changes the responsibility for that object from one logical process to another. This is called migration. The information previously saved during a *record object state* remains with the losing logical process. All the details of which object belongs to which sector and

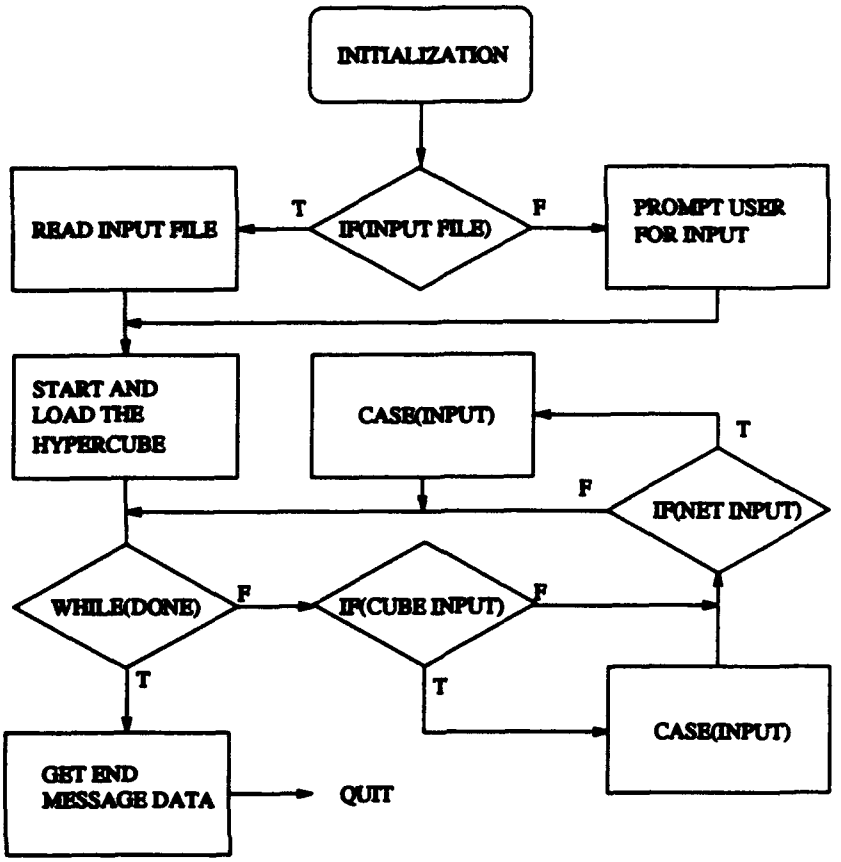

Figure 5. Visithost Flow Chart

to which logical process must be tracked and remain on the logical process that had the ownership at the time the *record* was made. Otherwise an effort to *restore* the state of a logical process would result in lost data since objects previously saved had migrated to other logical processes and were now resident in the memory of another node.

Presently this design implementation does not save all sector data. It is assumed that sectors will stay assigned to one logical process and not migrate from there. Boundary data on the sectors is saved to allow changes in the geographic responsibility of each logical process and thereby achieve load balancing for the parallel simulation.

The assumption is made that some higher level of the object hierarchy will remain located at one logical process throughout execution without migrating. The design of this research assumes

that the sectors will not migrate between logical processes. The assumption makes it easier on the design since it limits the amount of data that needs to be tracked and saved, but it is a false assumption. No level of the object hierarchy is required to stay attached to a single LP during the program execution.

In theory, the logical processes themselves could experience migration from physical node to physical node. This would require the state saving process to begin at the physical node level and save the state of all the logical processes resident on that node at the time of the SAVE STATE event.

4.2.4 Overall System Operation. The system begins execution through the initialization of a communication port over the network between the Visithost (on the host node of the Hypercube) and VISIT (on the SGI), (see Figure 3). The Visithost program sets up first, and waits for the handshake from VISIT. Once the port is open, both programs run through initialization. VISIT is ready and waiting for the *start display* message long before the Hypercube has been loaded with the BATTLESIM application. During BATTLESIM's initialization the start display message is generated and the user will begin to observe the objects moving once the first *object location* message arrives.

Each logical process in the simulation operates through a *while* loop that gets the next event off the next event queue and then executes it, (see Figure 6). If an *END* event occurs, the logical process waits for all the other processes to generate their END events and then the simulation ends gracefully.

During initialization the first *SAVE STATE* event is placed on the next event queue. When the logical process reaches this event, it calls the *record LP state* method to save the present state of the simulation. As with the END event, the logical process waits for the rest of the LPs to synchronize at this point before executing the SAVE STATE functions.

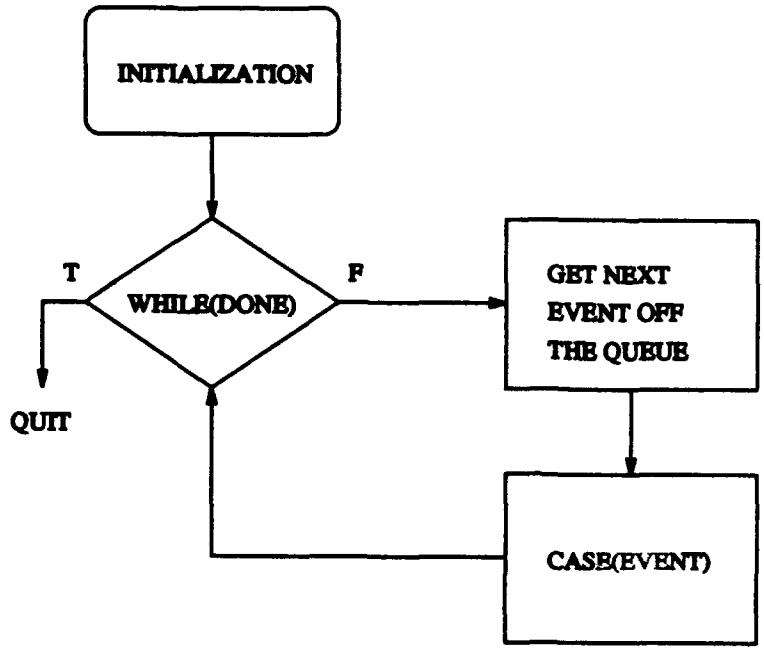

Figure 6. BATTLESIM Flow Chart

4.3 Extension Module Design

4.3.1 Purpose. In addition to the capabilities required for the simulation to save its state, the simulation designer must add the following extensions to perform user interaction during parallel execution. The first module encapsulates the functions required to format and transfer data to and from the simulation. The second module performs the synchronization and management of SAVE STATE events.

4.3.2 Use Visit. Use Visit is a collection of methods that allow a simulation to control the output of simulation data and interaction with the user. An output mode can be specified to either send update messages to a file, broadcast them across the network, both, or neither.

The choice of mode is selected by the user during initialization interaction with the Visithost program. The simulation is required to place the appropriate vehicle data (such as position, orientation, etc) in a routine call. The first item checked during a call to any of these routines is the

output mode. The Use Visit routine called then packages the data and makes the required transfer toward the network.

4.3.3 Sim Cntrl. Sim Cntrl is the collection of methods that allow the calling simulation to maintain a state saving capability. These methods require two things. First, the simulation must provide a means of tracking ownership of objects within each logical process. For example, the vehicles that are being processed by the logical process must be trackable. If the simulation does not provide this tracking then it will not be able to successfully store its state.

The second requirement is to provide the functionality described in Section 4.1.3 Once the simulation can store its own state, the *Sim Cntrl* methods can be used to keep the states in a time ordered list ready for restoration should the user request it. These routines also care for the synchronization of the state saving, storage of the state of the logical process, management of memory occupied by state data, and response to any user requested interaction with the simulation.

V. *Implementation and Testing*

5.1 Overview

The theory of an "easily" added extension for an interactive capability is relative. A candidate simulation for this modification will have to create the functionality specific to itself for the saving of its entire state. If the simulation currently uses an optimistic time synchronization protocol, this task should not be complicated. If the simulation has no state saving capability and the knowledge necessary to track the ownership of objects is hard to obtain, then the task is major.

The bulk of the research completed to demonstrate interactive control was performed on the simulation program. The implementation involved a simulation using a conservative time synchronization protocol and had no state saving capability. However, it did place an emphasis on object oriented design paradigms. The creation of the new methods was therefore straight forward, and the tracking of the object ownership among the LPs was easily exposed.

The balance of the tasks completed for this research were performed to complete system requirements such as a working link from the graphics workstation to the simulation. Some modifications were completed to satisfy unrequired enhancements, i.e. the VISIT interface was modified to be more user friendly and robust.

5.2 Design Implementation

5.2.1 Use Visit. As described in Chapter 4, the routines developed for the *Use Visit* capability serve the purpose of an interface for user interaction. Depending on the output mode selected at the start of the simulation, the routines allow for output to file, interaction on the network, both file and network use, or neither. The option of having no output was needed to record completion times for the simulation without input and output processes that would slow down the simulation execution. Each of the implementations of the routines are described below:

Print Something. This method is used to place a string in the output file (if that output mode is used). The research implementation uses this function to record the name of the terrain file used and the version number of the simulation.

Start Display. This method is used to notify the display program (VISIT) of incoming display messages. It is called at the beginning of the applications execution.

Stop Display. This method is used to cease communication with the display program and results in closure of the display file and/or last transmission from Visithost and Battoge. It is used at the end of the simulation.

Send Update. This method is used to transmit the next change in object location data to the display program. It is called by any application routine that enacts such a change.

Put Icon Id. This method is used to initialize the display program with icon identification numbers.

Put Icon Assn. This method is used to map the icon identification numbers to a icon representation such as a *mig, tank, truck, f16,* etc.

Put Obj Term. This method is used to notify the display program of the destruction of an object. It is called when an event such as a collision causes the termination of a perhaps a missile and an aircraft.

Init Use Visit. This method establishes the output mode as *graphic file* or *graphic net* or both.

Get Usr Cmd. This method checks the operating system input buffers for any commands that have been generated from the user for simulation control. It either generates a receipt message or calls the applicable routine to handle the command requested.

Check For Sim Cntrl. This method is used by logical processes when they have reached their END event. Since the simulation will usually finish execution before the user is done interacting with VISIT, a logical process must be dropped into a holding state before it goes

through the last few housekeeping chores and ends its execution. This method keeps the logical process from ending and waits to see if any further control commands come from the user. Only when the user selects the *quit* option from the display program will the simulation be allowed to complete.

5.2.2 Sim Cntrl. The implementation of *Sim Cntrl* works closely with *Use Visit* since the need to perform simulation control depends on the specified output mode. Implementation of synchronization functions, management of the simulation state, and performance of user initiated simulation control are the functions addressed by this package of routines. The specific routines follow:

Init Sim Cntrl. This method is called at the beginning of application execution. It retrieves logical process ownership information to decide which logical processes are the computational responsibility for which sectors. It also retrieves the output mode specified by the user. If the output mode does not specify use of the network, there is no reason to call any of the interactive simulation control methods.

Another important parameter retrieved is the time of the END event. There is a variable defined global to the simulation control object called *GRANULARITY*. This variable specifies the number of times that the simulation will save its state. The END time is divided by the GRANULARITY to produce a variable called *DELTA*. Every time a SAVE STATE event is encountered and the *record LP state* routine called, another SAVE STATE event is placed on the next event queue with a time of DELTA plus the current time. Finally, a linked list is initialized to hold state information produced later by SAVE STATE events.

Free State. This method is used by the *restore LP state* method to deallocate memory. This memory is associated with simulation states that are bypassed during the search for a state previous to the simulation time requested by the user for restart.

Synch LPs. This method simply sends a synchronization message to the Visithost program through an operating system call and blocks while it waits for the reply. When the Visithost program receives one of these messages with the same time stamp from each logical process it sends the GO AHEAD message back to all logical processes. Directly after receiving the GO AHEAD message, the logical processes execute the *get usr cmd* method from the *use visit* object. This access is synchronized to keep the logical processes from reacting to user simulation control commands independent of each other. Consider the following example described by Figure 7.

A possible problem exists if one logical process, LP 0, receives the STOP command before a synchronization point while the rest of the logical processes, LP x..z, receive the command after the synchronization point. LP 0 will react to the command about the same time LP x receives it. LP 0 will then be waiting for a RESTART command while LP x is waiting for LP 0 to arrive at the next synchronization point. If the RESTART is commanded, the synchronization will occur, execution will proceed, but when LP x executes the STOP command, LP 0 will continue its execution. Eventually the simulation will hang because the synchronization messages sent to the Visithost program arrive with different time stamps.

Set Cntrl Cmd. This method is called from the previously mentioned *get usr cmd* routine to set a flag in the simulation control object.

Get Cntrl Cmd. This method is called after the synchronization following record LP state routine. The execution consists of a case statement that checks for any user control command flags that may be set. Any action required is then initiated.

Record LP State. This method starts off the big state saving process. As mention in Section 4.1.3, this routine calls the top most routine in a hierarchy of routines provided by the simulation to save the simulation state. It keeps a reference pointer to the state and appends the state with a time stamp to a linked list of states. Here is the pseudo code for the routine:

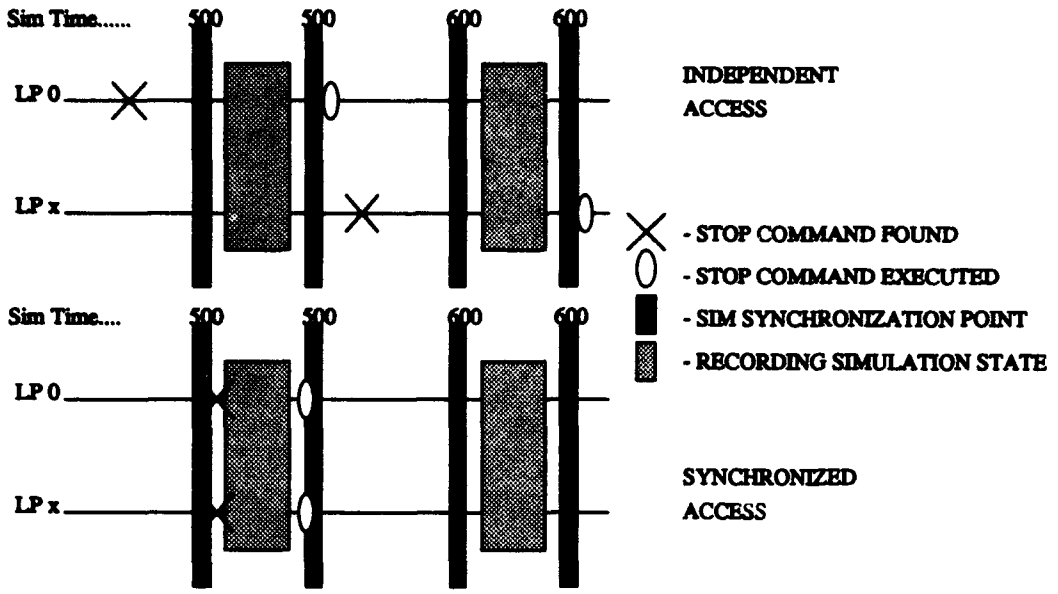

Figure 7. Possible problem with unsynchronized access to user commands

```
record_LP_state(simtime)
     - synchronize LPs
          -- set flag for any sim control command from the user
     - flush low level comm buffers and queues into NEQ
     - create new state element for linked list of states
     - assign time stamp to new state element
     - for each sector controlled by this LP
          -- create a new sector element for sector list
          -- save sector data
          -- save the set of objects in this sector
          -- for each object in the set, save its state
          -- insert sector element into list of sectors
     - assign sector list to new state element
     - save state of the next event queue
     - assign next event queue state to new state element
```

```
- insert new state element into the list
- create and enqueue next SAVE_STATE event
- synchronize LPs again
    -- set flag for any sim control command from the user
- check flags for any sim control commands from the user
    -- execute any commands present from the user
```

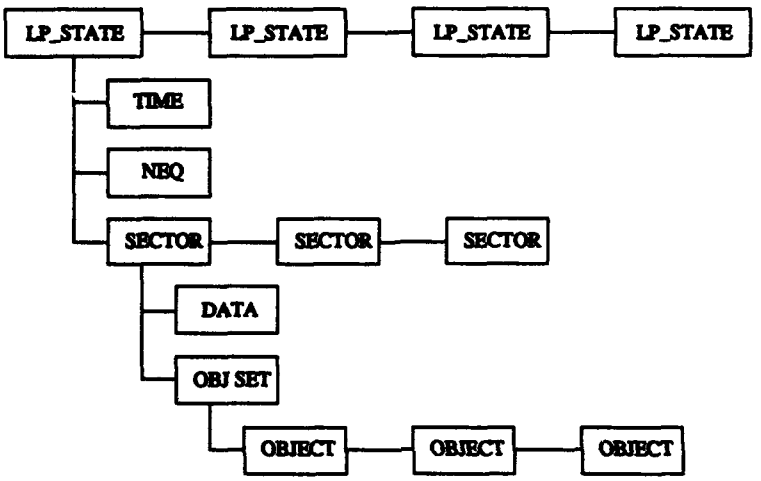

Figure 8. The shape of a saved state

Restore LP State. This method searches through the previously created list of states (Figure 8) until finding the appropriate time stamp. It frees all states recorded after the one selected and calls a hierarchy of routines provided by the simulation to restore its state with the new time.

Show State List. This method is used for debugging purposes to verify accurate state recording and restoring.

5.3 Implementation Details

5.3.1 State Restoring. The restoration of the simulation state in this design relies on the roll-back and coast-forward concept. The state seldom ever gets started from the exact position

requested by the user. Instead, the simulation restarts at the closest possible position prior to the requested time. Depending on the granularity variable, this could be acceptable or unacceptable. If the granularity variable is large then the number of states saved is large and the times between saves is small. When the user requests a roll-back the chances of getting close to the requested time increase if the time between saves is reduced.

5.3.2 Next Event Queues. When an event is retrieved from the next event queue, it must have the unique identifications of the objects involved in the event as well as a reference pointer to the locat n of the object's data. During a restore operation the reference pointers restored in the *restore object state* routine contain the actual pointers needed for update when the *restore next event queue* routine is called. Thus, the objects must be restored first and their reference pointers made available through some method when the restored next event queue needs the new copy. The unique identifications must be accurate and unchanged to obtain the pointers. Then when the next event queue is restored it can retrieve the newly restored object reference pointers through their unique identifiers.

5.3.3 Object States. The saving of object states is done through a *pack object* routine. This routine allocats a contiguous block of memory for each object and placs all of its data within that block. When the *restore object state* is executed, it calls the *unpack object* equivalent routine that placs the object back into its original form. Having the object in a packed form makes it very easy to *free*. Simply deallocating the memory associated with the reference pointer is all that is required. Otherwise the process because complicated, depending on the nature of the object structure.

5.3.4 Hierarchy and Migration. At some layer objects will have the capability of migration from the responsibility of one logical process to another. This forces a method for tracking the ownership of objects at layers that remain fixed. For instance, during initialization an array that

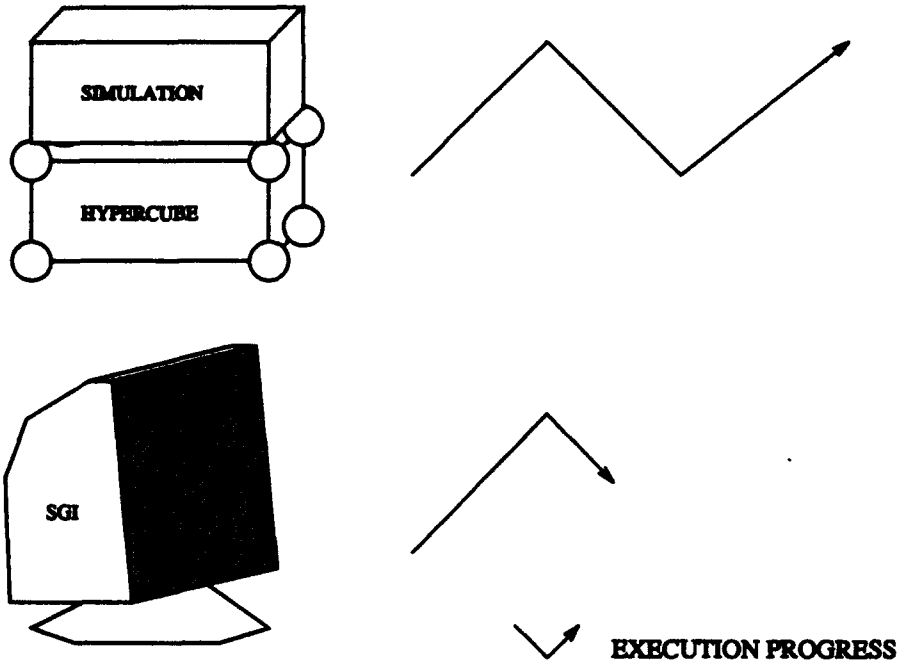

Figure 9. Display Lag

maps the logical processes to their respective sectors of responsibility is read and the information made available to the *sim cntrl* object (Section 5.2.2). This allows the *record* and *restore sector state* routines under each logical process to pursue only the states of sectors for which they are responsible.

5.3.5 The VISIT Buffer. Currently VISIT buffers in an array 200 object update messages for each object. The number 200 is changeable as a "define" statement and limited only by memory resources. The buffer exists to allow dead reckoning and to keep from losing data should the simulation be outperforming the display program as in the display lag example (see Figure 9). Enhancement has been completed to let this buffer begin rewriting itself from zero if the display program has displayed the data about to be overwritten.

The buffer's original intent was to allow the user to replay the simulation data from the graphics terminal. This will still be possible as long as the objects have not received more update messages than the number of locations available in the array. Since the successful roll-back of the

simulation is now possible, the user can replay the simulation itself rather than just the generated data. This implementation allows the user to be unrestricted by the size of the buffer, and the capability to modify objects in the simulation during execution.

5.4 Testing

The process of testing the products of this research followed the development in chronological order. Correct operation was validated for each of the following steps by printing status and format data to the console. Validation had to be achieved for each of the following links before progressing to the next step. This was both because a rigid software development practice was being followed, and because the next step couldn't exist without the previous one. Although a debugging tool was available for use on the cube nodes of the IPSC/2, it was not used. There is not a debugging tool for use on the host node of the IPSC/2.

VISIT to Visithost This link was established through TCP/IP network communications. A package of routines developed for the Air Force Institute of Technology by internal contractor support supplied all of the necessary functions. Reading and writing formatted data packets off of the ethernet connection between the graphics workstation and the host node of the IPSC/2 Hypercube was the extent of this requirement.

Visithost to BATTLESIM The IPSC/2 supplies operating system commands to perform communication between the simulation running on the cube nodes and the Visithost running on the host node. These calls provided all required parameters for development of unique messages. Both blocking and non-blocking calls could be made to check for incoming communication.

VISIT to BATTLESIM This link was tested as a function of the two previous links. For example, when the user initiated the SIM STOP command, a message was sent across the ethernet to the host node. It was taken from the network by a Battoge routine and passed to the

Visithost program. Visithost took the message in a buffer and sent it to all of the logical processes running on the cube nodes. The print statements showed up on the output screen of the simulation with a message that indeed the simulation had stopped. It then appears to hang while performing a blocked wait for the SIM RESTART command to be sent.

Redirect File Output The next step was to redirect the output information that had previously been sent to a file. The data was left in the same format but packed into a buffer and sent to the host node from the cube nodes via the operating system calls. The host node passed this buffer to a Battoge routine and from there onto the ethernet and the graphics workstation.

Implement Simulation Functions The development of the four methods described in Section 4.1.3 was next. The *record*, *restore*, *free*, and *show* routines were produced for each level of the simulation hierarchy. The validation for these routines was demonstrated by using the it show method. Output was sent to a file and inspected for correctness after the execution was finished.

User Interaction The last item was to implement the synchronization and perform the user interaction. This was produced correctly and observed on both the graphical workstation and the simulation output screen. The roll-back request caused the visible vehicle to return to a previous position and begin its route again, and the simulation output screen began repeating its own output messages as the execution started over from a previous state.

VI. Results and Conclusions

6.1 Results

6.1.1 Wins. The research was successfully completed for the development of requirements, design, and implementation of a remote interactive control capability for parallel simulation. The task was accomplished in a reasonable amount of time and can be applied to other parallel simulations by duplicating the requirements analysis and object oriented design described in Chapters three, four, and five. Several advantages can be realized by implementing this design.

- It can reduce the amount of memory resources needed to buffer the data at the remote workstation end of the system.
- The user may interact with the simulation during execution.
- The scope of the execution may be controlled.
- It can allow manipulation of objects or other simulation data.
- The advantages of roll-back are provided. A long simulation may be observed and altered without the need for restart from the very beginning.

6.1.2 Losses. Several possibly negative side effects could also impact the implementation of this design.

- Performance is reduced.
- Optimistic time synchronization protocols, like Time Warp (Section 2.5), lose their advantage due to the need for periodic synchronization of logical processes during a check for user input.
- Network communication adds complexity to the system through lag times and standardization of message formats.
- Due to state saving, the amount of memory resources at the simulation processor can suffer maximum utilization under extreme circumstances. If the number of SAVE STATE events is

large, the number of objects is large, and the amount of memory required to save the object is large, then the simulation could be limited by a small amount of memory. The Intel IPSC/2 does not support a function to easily check available memory during execution, so no data was generated to help quantify this limitation.

6.1.3 Performance. This section describes the variation in execution times against different output modes (to a file, across the net, both, neither). Different frequencies of SAVE STATE events are examined for the network mode. This data does not support different variations in the size and number of objects or the amount of memory usage per SAVE STATE event. All of these runs executed considerably faster than required by the graphics workstation. Any user interaction would not suffer simulation lag. To view the simulation at a reasonable pace took approximately one minute. None of the execution times came close to this amount of time.

Several factors need to be considered when analyzing the data in Table 1.

- Eight LPs were equally responsible for processing being done in sixteen sectors (two sectors per LP).
- Communication times over the network or between Hypercube nodes aren't deterministic. Even though the clock resolution on the Hypercube is 50ms, these times don't have a great deal of resolution partly because the error involved with network communication overcomes precise measurement.
- The user had to provide input over the network at the end of the simulation when using network mode. The only input was a QUIT command inserted right after the last SAVE STATE event was observed.
- The simulation was the only non-root process running during data collection. The IPSC/2 is a multi-user machine, but for these runs no other users were logged in. Some root processes did exist, so further error was introduced.

Table 1. Performance Data

Run Number	*Output Mode*	*Num SAVE Events*	*Wall Clock Time*
1	neither	N/A	4 sec
2	neither	N/A	3 sec
3	neither	N/A	2 sec
4	file	N/A	7 sec
5	file	N/A	7 sec
6	file	N/A	7 sec
7	both	10	15 sec
8	both	10	16 sec
9	both	10	16 sec
10	network	5	8 sec
11	network	5	8 sec
12	network	5	8 sec
13	network	10	10 sec
14	network	10	10 sec
15	network	10	10 sec
16	network	15	12 sec
17	network	15	11 sec
18	network	15	10 sec
19	network	20	13 sec
20	network	20	12 sec
21	network	20	12 sec
22	network	25	12 sec
23	network	25	13 sec
24	network	25	14 sec

- Eight LPs were partitioned across two nodes. This was an arbitrary decision, there is no reason the experiment was run on only two nodes versus four or eight. Each node has twelve megabytes of memory. Sixteen aircraft objects were being processed in this simulation, so the memory was not stressed. The aircraft were each within their own sector and did not perform any boundary crossings between sectors. In a simulation where many objects exist, it would be wise to partition more nodes per logical process to take advantage of more memory. The same argument applies to an increase in the number of times the simulation executes SAVE STATE events.

- Spreading the processing over all of the LPs equally minimizes the overhead spent executing a SAVE STATE event. Likewise, a simulation tasking a single LP with all of the processing maximizes the delay spent saving state. All of the other LPs must hold up and synchronize

with the processing LP, so in effect, the whole simulation is going to be dragged down by its weakest link.

The data suggests that increments of five SAVE STATE events only costs about one second. The major delay resulting from the network mode is from some other source besides the saving of the simulation state. Overhead such as processing of data structures to hold the saved state, or tasks performed when using the network is causing the delay. One loop executed only in network mode could create this extra delay. Such a loop is executed at the end of the simulation to make sure the logical processes all end gracefully together.

The file mode seems to incur small delay. Yet, when combined with the network mode the result is a delay of sixteen seconds. Six of these seconds are not accounted for since the data shows a delay of ten seconds with the network mode and the same granularity. Somehow the combination of both modes creates a bottleneck within the operating system to generate this extra delay.

6.2 Satisfied Requirements and Goals

6.2.1 VISIT. The VISIT tool met most of the requirements for this project. The only requirements driven change was implemented in order for the user to input a restart time. All other changes made were implemented to allow greater testability and user friendliness.

6.2.2 BATTLESIM. BATTLESIM was altered extensively to provide the required state saving and periodic synchronization capabilities. The required ability to accept user input during execution was also added. This function also required major additions to the governing routine running on the host node. The end product was the new routine called *Visithost* and a file called *Battoge* that contains all of the network access calls.

6.2.3 Interactive Control. The required interactive control capability involved the collective ability of the whole system. VISIT and BATTLESIM are now able to generate and receive

messages from each other. Battoge routines are used to do the decoding and encoding of network messages for BATTLESIM. This capability was more complex in this system due to the nature of the Intel IPSC/2. The host node has the only access to the network, so the simulation running on the cube nodes must have an intermediary running on the host to pass messages back and forth across the network. Regardless of the topology of the simulation computer, the interactive control capability must establish a synchronization point during access to the network.

6.3 Unsatisfied Requirements and Goals

6.3.1 VISIT. The only requirement left for VISIT to implement is a method with which to modify an entity's data and send the request back to the simulation. The user needs to be provided some input area(s) to access the data and directive prompts to help clarify the process. VISIT needs to be able to check the new data and start the process over if the user has placed the entity's data into an impossible state such as a submerged aircraft.

The VISIT tool has never been run against a simulation with large numbers of objects and multitudes of events per object. The VISIT buffer's current limit is 200 locations per object. This buffer would be in danger of overwrite if too many locations get generated before VISIT can display them to the user. The buffer has the capability to wrap around on itself and continue storing arriving locations, but it presently does not protect itself from overwriting un-displayed data. The capability exists now to check the progress of the display against available buffer space. When VISIT sees that the possibility of overwrite is approaching, the PAUSE command can be sent to the simulation in order to catch up. In this case, protection must be implemented to catch any user commands that are sent during the time the simulation has been paused. This capability does not yet work automatically.

While VISIT did satisfy most of its requirements, it is still lacking in the goals area. The user friendliness of the tool still needs to be improved. In the current implementation of VISIT the

user may either use the right mouse button for a pop up menu or a keyboard button to select a command. A friendlier interface may be implemented that uses a graphics window to display the simulation, and a control panel window with command options displayed more readily to the user.

The mouse could be used to perform selection of entities by placing the cursor over the entity and clicking a mouse button. The selected entity would be displayed with a unique identifier and other information on the control panel window.

Acceleration was not added to the dead reckoning capability performed by VISIT. This capability is needed in order for VISIT to achieve standardization with the new DIS protocol. Round earth modeling is also a requirement for DIS. Currently the graphics research group has successfully completed a round earth modeling algorithm. VISIT will need to interpret the data for a round earth model, if not perform the modeling itself. Further work toward improvement in both the acceleration and round earth problems should start with the graphics group's solutions.

VISIT requires simulation objects to be numbered sequentially starting at 1. This limits the flexibility and independence between the application and the interface. This problem should be an important priority on the next upgrade to VISIT. Refer to appendix D.

6.3.2 BATTLESIM. The requirements not yet implemented for BATTLESIM are involved with entity manipulation, mission manipulation, and tracking overlapping knowledge of entities when they are manipulated.

Checks need to accompany manipulation changes for the robust operation of the simulation. When the user generates a manipulation command, the validity of the request should be examined. Errors need to be handled gracefully if the object is non-existent, if two objects have been co-located, or if objects are placed outside their operational space. An acknowledgment from the simulation should respond to a selection by the user before execution is allowed to continue.

The coast-forward implementation is not yet present in BATTLESIM. The execution restart begins at the most recent state save performed before the requested roll-back time. In order to restart exactly at the simulation time requested by the user, the simulation must first roll-back to the most recent saved state, then schedule a RESTART event forward from that time. The simulation would then execute up to that event, and then query the user for further control commands.

6.3.3 Network Communication. The new Distributed Interactive Simulation network protocol is not yet implemented. This requirement will need a network specialist to first implement some package of calls that runs on the host node of the IPSC/2 for use by the Battoge routines. The transformation in protocols should be painless as long as the new calls match the form and function of the old calls. The Silicon Graphics workstation already has an implementation working for DIS protocols. This implementation is being used currently by the graphics research group. The UNIX V operating system running on the SGIs is similar to the Unix V operating system running on the IPSC/2 Hypercube. The designer of the SGI implementation suggested that it might work on the Hypercube with only a re-compilation. This theory is overly optimistic, but if the compilation errors can be solved then this avenue is certainly the most feasible. This research did not investigate the validity of these suggestions.

6.4 Conclusions

The demonstration of an interactive control capability for a parallel simulation was successful. The design and implementation of this research can be generalized for any candidate simulation to solve its interactive requirements. Control commands to stop, restart, pause, and continue the simulation can be implemented with a simple interface tool on a remote graphics workstation. Interaction to manipulate the objects within a simulation require a more complex interface. The complete knowledge of manipulated objects must be made available to the interface tool so that the user can administer changes that are consistent and realistic for the object within its environment.

Specific observations are detailed in the following subsections. These notes are meant to serve as advice for any research that concerns interactive control of a simulation.

6.4.1 Sequential Applications. The parallel simulation control capability can be easily scaled down to be applied to a simulation that runs on a sequential processor. Only the interface used to provide communication between logical processes needs to be altered.

6.4.2 Side Effects. Hardware that is developed to optimize the state saving process may be impacted by this design. The tracking of the simulation data and its possible migration during execution currently creates separate data copies referenced by pointer variables. Special hardware designed to get around such a brute force method will require a redesign to satisfy the tracking requirements.

6.4.3 State Saving. The DELTA interval separating the periodic saves will need to be optimized for several factors: the number of entities being saved, the mechanism used to save entity states, the amount of memory resources available to hold entity states, and the performance of the graphics workstation. The final consideration is important because of lag between the simulation and the display program. The simulation needs to keep ahead of the display program to prevent an incorrect display to the user, (see Figure 10).

The simulation's performance will be slowed if the number of entities is large, the interval is too small, the saving mechanism is slow, and/or the memory becomes constrained. If the simulation slows and cannot keep the graphics workstation supplied with current entity state data, the dead reckoning algorithm of the interactive graphics tool could extrapolate the position of entities outside their true course, causing the displayed icons to hop from a dead reckoned path back to their true path.

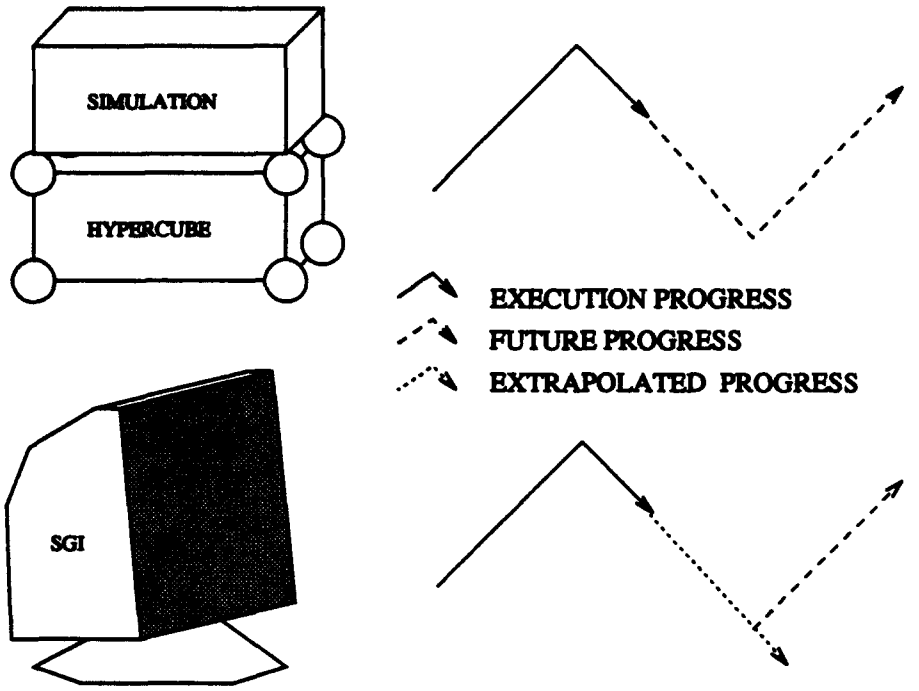

Figure 10. Simulation Lag

6.4.4 Lag and Roll-back. The situation may arise in which the display program extrapolates ahead of the simulation (Figure 10). In an extreme case this may allow the user to stop the program and issue a RESTART command that has a time stamp greater than the simulation time yet reached by the simulation itself. This is an error condition as yet un-encountered, but something to check for under extreme circumstances.

6.4.5 Time Synchronization Protocols. The use of an optimistic time synchronization protocol with this design may be easier to achieve because the state saving functionality is already present. However, the speedup gain achieved by letting logical processes execute without waiting for possible input may be reduced by the need to periodically synchronize the processes to check for user input. Depending on the granularity variable, (how often the logical processes synchronize to save state and check for user input), the logical process that would have run far toward early completion would have to stop and wait every DELTA time units.

6.4.6 Manipulation of Objects. This research project did not attempt to modify objects during the simulation execution. Neither the parallel simulation research faculty nor the sponsor of this research required this ability. However, in order to achieve this in the future, the simulation itself must provide the means to not only modify the object parameters itself, but the means to track where copies and knowledge of the object exists. This must be done so that the object is updated across the simulation, not just local to the logical process that is the current owner of the process. For instance, a radar tracking object sensing an aircraft will have to be updated with a new position if that aircraft is moved around.

Research has been completed in parallel with this project that now provides the capability to perform object manipulation. A redesign of the BATTLESIM simulation now allows a greater capability in tracking of objects across the simulation. See reference (22). The next step is to provide the VISIT interface tool with the ability to alter an object's complete state. VISIT must be able to update an object's position, route, velocity, acceleration, and sensing parameters. Changing the state of one object during the execution of the simulation will also require the rest of the objects in the simulation to be updated with the new information. This is the knowledge problem described in the above paragraph.

6.4.7 Granularity vs Resources. The user will have to be aware of the memory resources available to save the state of all objects in the simulation. This is one of the purposes of the granularity variable. If the number of periodic saves is small (the granularity is small), then the amount of memory needed is reduced. The user can also maximize the resources available to each logical process by loading one logical process onto one physical node of the parallel computer, rather than several logical processes per node.

6.4.8 Optimization. The simulation loses performance every time it pauses to synchronize and save state. This can be minimized by a small granularity, simple objects, fewer objects, and a speedy algorithm for state saves. Performance is required to keep the problems described in

the simulation lag example from happening (Figure 10). Data should be compiled to compare the simulation time for each output mode and different granularities.

Appendix A. Simulation Developer's Guide

Introduction

This appendix is meant to assist anyone who will be using the VISIT interface that has to design into a simulation the various "hooks" used by this interface. There are various features that should be incorporated into a simulation in order to utilize the maximum functionality of this interface. However, very little extra work should be required to just view a simulation; the majority of the extra work is required for viewer interaction and control.

The *Visithost* program should be used to with a "-n" option entered at the command line if the user wishes to output messages across the network. Actually, the Visithost program just checks to see if any command lire arguments are present. As long as the number of options is greater than one Visithost will try to synchronize with VISIT over the ethernet.

Visithost also checks to see if the "visithost.inp" file is present. This file holds all of the options needed to load the Hypercube. If the file is not in the same directory as the Visithost executable, then the program will interactively prompt the user for these options.

New options to input with BATTLESIM include *-o* and *-g*. The *-o* option should be followed by a letter to specify the output mode, default is file mode:

n or N for network mode.

f or F for file mode.

b or B for both modes.

z or Z for neither mode.

The *-g* option should be followed by an integer to specify the number of times the user wishes the simulation to save its state during execution. Default is ten.

Data Representations

The objects within the simulation are represented graphically on the display system by *icons*. To represent an object on the display screen the graphics display system must know what to use for its representation. These icons are described using polygonal descriptions in the file format specified in Appendix F.

All objects are described with their center oriented at the origin of the world coordinate system. The front of the object is pointed away from the origin down the positive y axis. The initial orientation of the object is zero for all rotation angles. All objects are measured in meters and are externally scalable to any size. The interface allows the viewer to control the object scale size, but that value applies to all objects known to the display system.

The orientation angles are measured according to the right-hand rule, which is as follows: as you look down the positive rotation axis to the origin, positive rotation is counterclockwise. The interface reads a parameter at initialization time that may be used to indicate that the angle measurements are the reverse of this rule. The important thing to remember is be consistent, either all angles are counterclockwise positive or clockwise positive.

The position of objects are specified using the 3-dimensional right-handed cartesian coordinate system. The units used are of no importance to the display system except for consistency. Whatever unit of measurement is used for positioning, must be used throughout. If you use meters to describe the objects and miles to indicate their positions, an erroneous display will occur. Also, there is a coordinate scaling parameter in the parameter file that may be used to indicate to the display system to scale all coordinates by this factor.

When using a unit of measure other than meters, the object descriptions may be scaled within the interface by the viewer. A keyboard function is available to the viewer to scale up or down by a factor of 2 all object representations. This is a global factor that applies to all objects and may not be applied to individual objects.

Object positions and orientations are sent to the graphics display system whenever a position or orientation changes. The packet that is sent requires the position coordinates, orientation angles, and velocity vector coordinates for the position and orientation. This information is used to determine the object's movement within the simulation. These records must be sent to the display system in simulation time order for any given object.

Communications

Appendix D describes the various message packets that are sent back and forth between the two systems. Appendix C describes the format of the datafile used for data mode operation by the simulation postprocessor, Simtoge. Simple playback viewing of a simulation can be accomplished by creating a datafile in the format specified and using the data mode of operation. However, to utilize the full functionality of the interface direct communications between the simulation processor and the graphics display system is required in real-time.

The necessary modules required for communications with the display system are available in the library /usr/local/lib/AFIT_comc.a. This file contains the network communications modules used by the interface and the simulation postprocessor, Simtoge. The modules also contain the message packet structure that is used to transfer the data between the systems.

Many simulation control capabilities are provided by the interface that rely on the simulation processor to perform some function. In data mode these message packets are built and sent to Simtoge and an acknowledgment message is displayed in the console window. The viewer will have the ability to send the simulation a message to pause, stop, continue, restart, or abort the simulation. Also, the display system will send a message to the simulation processor when the display system reaches a viewer established breakpoint.

To assist the simulation in communicating with the display system, display status messages exist. The simulation must send a packet to the graphics display system when it is ready for the

display system to begin displaying. This allows the simulation to perform all of its initialization, identify its objects, and begin its execution before the display system begins.

It is important to remember that the display system will continue to display and update the simulation objects based on what is currently known. This requires the simulation processor to keep ahead of the graphics display system with respect to simulation time. The viewer may control this by pausing the display. Pausing the display pauses the display of the simulation at the current simulation time. In the background communication will continue between the two systems, thus allowing the simulation processor to continue sending data.

Finally, when the viewer exits the interface system a message packet is sent to the simulation processor to terminate. This is a send-only request, no return message will be accepted.

Viewer Control & Interaction

The viewer also has the ability to relocate and delete objects that are currently known to the graphics display system. Packets will be sent to the simulation processor when either of these actions are taken by the viewer. This will require some action be taken by the simulation. When an object is deleted by either a termination record from the simulation or a delete object from the viewer, the object number may not be reused.

Summary

The information presented in this appendix is intended to be a supplement to the other portions of this thesis. It is highly recommended that a developer read Derouchey's work, all appendicies to it, and this document before attempting to develop a simulation that will require the VISIT interface.

Appendix B. VISIT User's Guide

Introduction

VISIT is an interactive simulation display system that permits the graphical display of a simulation that is concurrently executing on another system. Display data is received by the graphics display system and commands are sent to the simulation processor via ethernet communication utilizing the TCP/IP protocol.

The following sections explain how to setup and operate VISIT. In addition, the various file formats are explained in detail.

Preparing VISIT for Use

Before VISIT can be used various initialization tasks must be accomplished in the proper sequence.

The following procedure should be used when executing the display of a simulation.

1. Begin by logging into the Silicon Graphics Iris 4D/85GT
2. Ensure appropriate files are present on the systems used (see Table 2).
3. Type VISIT <cr> to execute the main menu
4. Select the appropriate configuration from the menu.

Once the message to start displaying the simulation is received, the graphics system will begin the display. The display initially appears in pause mode to allow the viewer to manipulate

Table 2. Data Files Required

File	*Description*
configuration file	identifies spaceball configuration (if using)
parameter file	identifies initial viewing parameters (params.dat)
data file	simulation records datamode only
rendering executable	gprfilter

the viewing parameters or to change the display control characteristics. The animation of the simulation will begin after a keyboard p is pressed. To exit the system, the viewer enters a keyboard q. An on-line help facility is available by typing HELP <cr> in the help window.

Interactive Input

The parameter file provides the various initial parameters. However the system provides the flexibility to interactively modify the various viewing parameters and the simulation objects themselves. Keyboard functions, a mouse menu, and the spaceball functions provide these capabilities.

Keyboard & Mouse. The keyboard provides access to virtually all viewing parameters and allows the user to interactively modify the simulation environment. The list of keyboard functions is displayed in Table 3.

The mouse provides a single popup menu by clicking on the right mouse button. The options are selected by clicking the right mouse button when the applicable option is highlighted. To cancel before selecting an option, click on the menu title and you will return to the display. The popup menu functions are listed in Table 4.

CIS Dimension Six Trackball. This device has not been used with VISIT since DeRouchey. Improvements in VISIT to allow greater flexibility with the mouse should overcome the need of any special device for picking and moving objects.

THE CIS Dimension Six force-torque ball is a six degree of freedom input device combining the functionality of a joystick, a button box, and a dial box. The ball allows rotations about the three primary axes and translations in three directions. Force sensors inside the ball register the amount of force and torque applied to the ball and send this information over an RS-232 interface to the host.

Table 3. Keyboard Functions

Key	*Function*
+	increase field of view
−	decrease field of view
b	toggle spaceball
c	toggle clock display
f	decrease time scale
F	increase time scale
g	toggle terrain grid
l	toggle cockpit mode
p	toggle pausing display
q	quit the simulation
r	reset clock to 0.0
s	decrease scale factor
S	increase scale factor
t	toggle terrain display
v	toggle viewing parameters
w	toggle wireframe mode
x	toggle display trails

Table 4. Mouse Menu Functions

Option	*Function*
Toggle Clock	toggle clock display
Toggle Parameters	toggle viewing statistics
Toggle Terrain	toggle terrain display
Toggle Pause	toggle pausing display
Simulation Control	submenu to control simulation
Clock Control	submenu to control display clock
Pick Object	pick object to view from
Change Terrain	change terrain file
Pause	send msg to pause simulation
Continue	send msg to continue simulation
Stop	send msg to stop simulation
Abort	send msg to abort simulation
Restart	send msg to restart simulation
Reset Clock	set clock to 0.0
Set Clock	set clock to time n
Set Breakpoint	set breakpoint to pause display
Step Display	step through display by frames

Table 5. Trackball Function Keys

Key	*Function*	*Comments*
1	Display Cube	Begin picking mode
2	Pick Object	
3	Cancel Pick	
4	Drop Object	
5	Activate Zoom	
6	Delete Object	
7	Activate Pan	
8	Deactivate	Terminates movement

The device has eight function buttons that provide user input capability (see Table 5). There is also a set of three buttons on the trackball that allow the user to change from translations to rotations. The force applied to the ball is sent to the host and read into a data structure. The status of the three control buttons determines how that data structure is filled. The trackball is used to both modify the viewing parameters and alter the location and/or existence of simulation objects.

Viewing Mode. In viewing mode the user is allowed to either zoom in or out, or pan along one of the three primary axes. To zoom, the user depresses a function button and then applies force to the ball in the direction specified in Table 6 to achieve the desired change in the display. As the system reads the amount and direction of force, the display is updated. Once a position is reached and the user is satisfied, a function button is depressed to exit viewing mode.

To pan the display, the user depresses a function button and then applies force to the ball in the direction in which the display is to pan. This system uses the left-hand coordinate system, so applying force to the right will pan along the positive x axis, pushing forward on the ball will pan along the positive y axis, and pulling up on the ball will pan along the positive z axis. Applying force in the opposite direction for either axis will pan along the respective negative axis. Again, once a suitable position is reached, a function button is depressed to exit viewing mode (see Table 5).

Table 6. Trackball Translations

Key	*PAN Function*	*ZOOM/MOVE Function*
slide ball forward	pan forward	move entity along +y
slide ball backward	pan backward	move entity along -y
slide ball right	pan eastward	move entity along +x
slide ball left	pan westward	move entity along -x
slide ball upward	pan upward	move entity along +z
slide ball downward	pan downward	move entity along -z

Picking Mode. The user can maneuver a cursor anywhere within the current display screen and 'pick' an object, move that object with the trackball, and 'drop' the object at the new location. The user may also delete an object from the current display once it has been picked. If the object that the user wants to pick is not currently within the viewing window, the user can enter viewing mode and reposition the window so that the desired object is viewable.

The user enters picking mode by depressing a function key on the trackball. This causes the system to display a wireframe cube in the center of the viewing window. The user then maneuvers the cube by applying force in the appropriate direction towards the object that is to be picked (see Table 6). When the object appears within the cube or near it, the user depresses another function key. This triggers the software system to calculate the distance from the cube to all viewable objects within the simulation. The closest object to the cube is then displayed in wireframe mode for visual feedback to the user. At this point the user has three options.

If the object that was calculated by the system to be nearest to the cube is *not* the object that the user wanted selected, a function key is available to 'unpick' the object. This will reset the display mode of the object back to shaded and display the cube again. The user may then maneuver the cube closer to the desired object and pick it again.

A second option after the object is picked is deletion. The user can depress a function key that will cause that object to be removed from the current list of displayable objects. This will also trigger the system to send a network message to the simulation system that an object was deleted from the simulation.

A final option is to relocate the object. Once the object has been picked, the user may apply force to the ball and thus relocate the object within the viewing window. The user can also reorient the object by applying force to the ball in any one of three rotation directions. The user can switch between translations and rotations by depressing the applicable control button on the device. Once a suitable position and orientation has been achieved, a function key is depressed to 'drop' the object at its current displayed location. This also triggers the system to send a network message to the simulation that the location of an object has been altered.

Appendix C. Datamode File Format

The datafile is composed of records of several types. Each record type contains fields in a specific format. The number of fields in a record is different for each record type. In all cases, the first field contains an integer which defines the record type.

Types

Icon Assignment. Assigns an icon index to a viewable object.

Example: `30 3 8`

This examples indicates simulation object number 3 is assigned to icon index 30. Simulation Object numbers must begin with 1 and be sequential.

Object Location. Contains position and orientation data for a viewable object. The position and velocity values have a maximum width of eleven characters. This width is inclusive of a minus sign and a decimal position. The angles are measured according to the right-hand rule, which is as follows: as you look down the positive rotation axis to the origin, positive rotation is counterclockwise.

Example: `31 2 2.5 1000 500 -20 1.2 2.4 -.3 30.0 10.0 -90.0 0.5 5.0 -1.0`

This examples indicates that object number 2 at simulation time 2.5 is positioned at 1000, 500, -20. The object has a heading of 30 degrees, a pitch of 10 degrees, and a roll of -90 degrees. The velocity components are 1.2, 2.4, and -.3 for position and 0.5, 5.0, and -1.0 for orientation.

Table 7. Record Type 30

Field	*Description*	*Data Type*
30	Record Type	integer
O	Object Index Number	integer
I	Icon Index Number	integer

Table 8. Record Type 31

Field	*Description*	*Data Type*
31	Record Type	integer
O	Object Index Number	integer
T	Time (seconds)	float
X	X - position (meters)	float
Y	Y - position (meters)	float
Z	Z - position (meters)	float
VX	velocity in x (meters/sec)	float
VY	velocity in y (meters/sec)	float
VZ	velocity in z (meters/sec)	float
H	Heading (degrees)	float
P	Pitch (degrees)	float
R	Roll (degrees)	float
VH	change in Heading (degrees/sec)	float
VP	change in Pitch (degrees/sec)	float
VR	change in Roll (degrees/sec)	float

Table 9. Record Type 32

Field	*Description*	*Data Type*
32	Record Type	integer
I	Icon Index Number	integer
F	Icon Filename	character string

Icon Identification. Identifies an icon by index and geometry description filename.

Example: 32 8 mig1

This example indicates that icon index number 8 is associated with the geometry file mig1. Icon numbers are determined freely by the user.

Object Termination. Identifies when an object is to be terminated. This is the time at which the display system stops displaying the object.

Table 10. Record Type 33

Field	*Description*	*Data Type*
33	Record Type	integer
O	Object Index Number	integer
T	Termination Time	float

Table 11. Record Type 50

Field	*Description*	*Data Type*
50	Record Type	integer

Table 12. Record Type 52

Field	*Description*	*Data Type*
52	Record Type	integer

Example: `33 3 115.5`

This examples indicates that object number 3 will be no longer displayed at simulation time 115.5.

Start Display. Indicates all icons and the initial starting positions have been identified and sent to the graphics engine. The graphics engine can begin displaying the simulation.

Reset Display. Indicates to the graphics display system that the simulation was restarted and will begin execution. The graphics display system will pause until a START DISPLAY is received.

End of Simulation. Indicates the end of the simulation. This will be the last line within the datafile that is read.

Example: `86 245.0`

This example indicates that the simulation display is to stop at simulation time 245.0.

Table 13. Record Type 86

Field	*Description*	*Data Type*
86	Record Type	integer
T	Termination time	float

Ordering

All icon identifications (type 32) must occur before any other type of record in the datafile. Each viewable object must be associated with an icon (type 30) before a location record (type 31) for that object can occur in the datafile.

Appendix D. Message Packet Descriptions

Introduction

The fields of the message packet used by this interface are shown in Table 14. The icon field is only used by the icon assignment and icon identification type packets; all other packets avoid this field. The buf field contains type-specific data and is identified specifically for each type packet.

This message packet structure was adapted from an existing network communications package. This package allowed communication between the 4D/85GT system and a parallel processing system; however, it relied on the message header fields be of type short. This restriction prevented an easy way of storing the simulation time, (a float type), within the message header, and that is why the simulation time is stored in the buffer instead of being a field within the message header.

Data Types

Table 14. Message Packet Structure

Field	*Description*
request	Request Type
object	Object Index Number
icon	Icon Index Number
bufIth	Length of Buffer
buf	Data

Type 30 - Icon Assignment. Assigns an icon index to a viewable object. The buffer contains 0 bytes of data for this record. Object numbers must begin with 1 and be sequential.

Type 31 - Object Location. Contains position and orientation data for a viewable object. This record type should not be sent until after the icon assignment packet has been sent for the object number in this packet. The buffer contents for this record are the following:

simulation time,x,y,z coordinates, x,y,z velocity vector coordinates, heading, pitch, roll, heading change, pitch change, and roll change.

All values are separated by one space character.

Type 32 - Icon Identification. Identifies an icon by index and geometry description file that is used by the display system to represent the object. The buffer contains the filename only. This filename is concatenated to the path name provided in the parameter file. This packet type must be sent for each icon used, but before any icon assignment is sent for that particular icon.

Type 33 - Object Termination. Identifies the simulation time in which an object is terminated. The buffer contains the simulation time at which termination occurs.

Viewer Interaction Types

These message packet types are used when the viewer interacts with the display of the simulation using the spaceball device. These commands have no effect in the data mode of operation.

Type 34 - Move Object. This message packet is used to send the new location of an object back to the simulation. The contents of the buffer are: simulation time, x, y, z position, roll, pitch, and heading.

Type 35 - Destroy Object. This packet is used to send the simulation time at which the user interactively deleted an object. The simulation time is placed into the buffer.

Simulation Control Types

The simulation control is quite different from the display control capability of the system. When the user selects to pause the display only the display and object position updating pause; no message packet is sent to the simulation processor. The simulation control packets are for communication with the simulation processor. The current simulation time is loaded into the buffer before the packet is sent for all packets of this type.

Type 40 - Pause Simulation. The display system sends this message to the simulation system when it becomes necessary to pause the simulation. Currently, this condition never occurs internally and is allowed only as a user selected option.

Type 41 - Continue Simulation. The display system sends this message to the simulation system when it becomes necessary to continue the simulation. Currently, this condition never occurs internally and is allowed only as a user selected option.

Type 42 - Stop Simulation. The display system sends this message to the simulation system when it becomes necessary to stop the simulation. Currently, this condition never occurs internally and is allowed only as a user selected option.

Type 43 - Abort Simulation. The display system sends this message to the simulation system when it becomes necessary to abort the simulation. Currently, this condition never occurs internally and is allowed only as a user selected option.

Type 44 - Restart Simulation. The display system sends this message to the simulation system when it becomes necessary to restart the simulation. Currently, this condition never occurs internally and is allowed only as a user selected option. This is the only simulation control message packet that the simulation postprocessor, Simtoge, uses. The other five packets are read and an acknowledgment is displayed in the console window; no action is taken.

Type 45 - Save State. The display system sends this message packet when a user determined breakpoint is reached.

Type 25 - Quit Communications. This packet is sent by the graphics display system as a house-cleaning function before termination of the interface. The buffer contains the current simulation time.

Status Types

Type 50 - Start Display. Instructs the graphics display system to begin displaying the simulation. At this point the display system will allow user interaction and control. This packet should not be sent until all icon identification and icon assignment packets have been sent for the objects that are known at simulation time zero. The buffer and object index field will be empty.

Type 52 - Reset Display. Instructs the graphics display system that all current data is no longer valid and the simulation will restart. The simulation must send a Start Display packet to allow the display system to continue. The buffer and object index field will be empty.

Type 86 - Datafile Complete. Instructs the graphics display system that reading of the datafile is complete. The buffer and object index field will be empty.

Modifications

As stated in the introduction this message packet structure is based upon a communications package received from Oakridge National Laboratory. The modifications that were made are noted here because it's the only place that they fit in.

First, the original package relied on a global visibility for the network socket file descriptor and message packet structure. That could not occur in the interface. It was necessary to make these two entities parameters for all modules that required them.

Second, several new request types were added to the original package. Basically all request types identified in this appendix are new and are identified in the header file /usr/local/include/AFITcom.h. All previous request types were maintained. A subset of the original collection of modules was combined into one file, identified as /veds/oakridge/lib/simutils.c. This file is linked in by both the interface VISIT and the simulation postprocessor Simtoge.

Appendix E. Parameter File Format

The VISIT parameter file provides a method for initializing the viewing parameters. The file is arranged so that only one parameter keyword and its associated value(s) can be on a single line. The order in which the parameters appear within the file is not important.

The file syntax is shown in Table 15. In the table, literal symbols appear as contiguous strings of alphabetic characters. Substitutable symbols appear in angle brackets "<" and ">". Optional information appears within square brackets "[" and "]". C-style comments (`/* */`) and `#include` and `#define` directives may be contained within a parameter file.

Table 15. VISIT Parameter File Format

Keywords	*Comments*
COI <x> <y> <z>	Center of Interest Coordinates
EYE <x> <y> <z>	Eye Position Coordinates
FOV <angle>	Field of View
FUR <rate>	Frame Update Rate
IDIR <path>	ICON Directory
MTT <time>	Maximum Simulation Time
RA <angle>	Rotation Angle Amount
SF <factor>	Object Scaling Factor
SS <factor>	Movement Step Size
TER <x> <y> <z>	Terrain offset
TF <factor>	Time Scaling Factor
TFILE <filename>	Terrain Geometry Description
CSCALE <factor>	Coordinate Scaling Factor
MANGLE <mode>	Angle Measurement in CW Direction
NPLANE <distance>	Near Clipping Plane
FPLANE <distance>	Far Clipping Plane

Appendix F. AFIT Geometry File Format

The AFIT geometry file provides a method for describing three dimensional objects composed of planar polygons, and for specifying attributes such as color and a shading model. The file is organized in a position dependent manner such that the position of a line within the file (its "line number") determines the class of information a line may contain.

The file syntax is shown in Table 16. In the table, literal symbols appear as contiguous strings of alphabetic characters. Substitutable symbols appear in angle brackets "<" and ">". Optional information appears within square brackets "[" and "]". C-style comments (`/* */`) and `#include` and `#define` directives may be contained within an AFIT geometry file.

Table 16. AFIT Geometry File Format

Line	*Keywords*	*Comments*
1	None	Up to 1024 characters
2	[ccw] [cw] [purge] [nopurge]	Geometry Parameters
3+	points <# of points> patches <# of patches> [attributes <# of attributes>] [textures <# of textures>]	Object component and attribute counts
4+	<x> <y> <z> [normal <i> <j> <k>] [color <r> <g> <b> [tindex <u> <v>]	Vertex Lines
5+	<n> <pt 1>...<pt n> [attribute <n>] [texture <n>] [type {PLAIN, COLOR, TEXTURE}]	Polygon/Patch Lines
6+	[shading {FLAT, GOURAUD, PHONG}] [reflectance {FLAT, PHONG, COOK}] [kd <n>] [ks <n>] [n <n>] [opacity <n>] [color <r> <g> <b>] [m <n>] [material <filename>]	Attribute Lines
7+	filename	Texture Map Filename

Bibliography

1. ASC/RWWW. *System/Segment Specification for the Joint Modeling and Simulation System Program*, September 7 1992. Version 1.1.
2. Bergman, Kenneth C. *Spatial Partitioning of a Battlefield Parallel Discrete Event Simulation*. MS thesis AFIT/GCS/ENG/92D-03, Air Force Institute of Technology, 1992.
3. Bischak, Diane P. and Stephen B. Roberts. "Object-Oriented Simulation." *Proceedings of the 1991 Winter Simulation Conference*. 194–203. New York: IEEE Press, 1991.
4. Chandy, K. M. and J. Misra. "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Communications of the ACM*, *24*(11):198–206 (May 1981).
5. Chandy, K. M. and Jayadev Misra. "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, *SE-5*(5):440–452 (Sep 1979).
6. Chandy, K. M. and R. Sherman. "The conditional event approach to distributed simulation." *Proceedings of the SCS Multi-conference on Distributed Simulation 21*. 93–99. Mar 1989.
7. DeRouchey, William J. *A Remote Visual Interface Tool for Simulation control and Display*. MS thesis AFIT/GCS/ENG/90D-03, Air Force Institute of Technology, 1990.
8. Fujimoto, Richard M. "Parallel Discrete-Event Simulation," *Communications of the ACM*, *33*(10):31–53 (Oct 1990).
9. Haddix, Rex G. *An Immersive Synthetic Environment for Observation and Interaction with a Large Volume of Interest*. MS thesis, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, AFIT/GCS/ENG/93M-02, March 1993.
10. Hartrum, Thomas C. *AFIT Guide to SPECTRUM*. Unpublished Report, Air Force Institute of Technology, Feb 1992.
11. Hartrum, Thomas C. *TCHSIM: A Simulation Environment for Parallel Discrete Event Simulation*. Unpublished Report, Air Force Institute of Technology, Jan 1992.
12. Intel Corporation. *iPSC/2 User's Guide*, Mar 1989. Order Number: 311532-003.
13. Jefferson, David R. "Virtual Time," *ACM Transactions on Programming Languages and Systems*, *3*(7):404–425 (Jul 1985).
14. McDonald, Bruce. *Distributed Interactive Simulation*. Development Guidance Document(DRAFT), University of Central Florida Institute for Simulation and Training., 1992.
15. Reynolds, Paul F. "A Spectrum of Options for Parallel Simulation." *Proceedings of the 1988 Winter Simulation Conference*. 325–332. Dec 1988.
16. Reynolds, Paul F. "Comparative Analysis of Parallel Simulation Protocols." *Proceedings of the 1989 Winter Simulation Conference*. 671–679. Dec 1989.
17. Reynolds, Paul F. "SRADS with Local Rollback." *Proceedings of the SCS Multi-conference on Distributed Simulation 22*. 161–164. Jan 1990.
18. Rizza, Robert J. *An Object-Oriented Military Simulation Baseline for Parallel Simulation Research*. MS thesis AFIT/GCS/ENG/90D-12, Air Force Institute of Technology, 1990.
19. Sheasby, S. M. *Management of SIMNET and DIS entities in synthetic environments.*. MS thesis AFIT/GCS/ENG/92D-16, Air Force Institute of Technology, 1992.
20. Soderholm, S. R. *A Hybrid Approach to Battlefield Parallel Discrete Event Simulation*. MS thesis AFIT/GCS/ENG/91D-23, Air Force Institute of Technology, 1991.

21. Switzer, John C. *A Synthetic Environment Flight Simulator The AFIT Virtual Cockpit.* MS thesis, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, AFIT/GCS/ENG/92D-17, December 1992.

22. Trachsel, Walter G. *Object Interaction in a Object Oriented Parallel Discrete Event Simulation.* MS thesis, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, AFIT/GCS/ENG/93D-22, December 1993.

23. Van Horn, Prescott J. *Development of a Protocol Usage Guideline for Conservative Parallel Simulations..* MS thesis AFIT/GCS/ENG/92D-19, Air Force Institute of Technology, 1992.

Vita

Douglas C. Looney was born in Ontario, Oregon in December of 1966. He was educated until the age of eighteen in Ontario. He graduated from Ontario Senior High in the spring of 1985. Having earned a Reserve Officer Training Corps scholarship, Doug left Ontario to complete an engineering degree from the University of Portand, Oregon.

Looney was awarded the degree of Bachelorate of Science in Electrical Engineering by UoP on 30 April 1989, one day after the Air Force commissioned him as a Second Lieutenant.

Lieutenant Looney's first assignment was at Wright Patterson AFB in Dayton, Ohio. It was here that he was trained to acquire software systems for the Air Force. Looney was assigned to two missile programs in a row as a software engineer. Both the Joint Tactical Anonymous Weapons (JTAW) and Short Range Attack Missile II (SRAMII) System Program Offices were closed during his service to them. Looney left Aeronatical Systems Division for AFIT in the spring of 1992.

Looney has been reassigned to AFOTEC at Kirtland AFB. His organization symbol is HQ AFOTEC/SAS. Email dlooney@hq.afotec.af.mil.

Permanent address: 1642 Betts Ct, NE
Albuquerque, NM 87112

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1993	Master's Thesis

4. TITLE AND SUBTITLE

Interactive Control of a Parallel Simulation

5. FUNDING NUMBERS

6. AUTHOR(S)

Douglas C. Looney, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

CAPT RICK PAINTER
2241 AVIONICS CIRCLE, SUITE 16
WL/AAWA-1 BLDG 620
WPAFB, OH 45433-7765
513-255-4429 or DSN 785-4429

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Modern military commanders are faced with an overwhelming amount of intelligence data concerning the disposition of engaging forces. The sheer volume of data produced for a single planning scenario is an obstacle to the user as well as the computer.

Today's commander requires a real-time, three-dimensional representation of the battlefield in order to asymilate the data for the management of a conflict. Parallel computation is required to complete the processing of this information in a timely manner.

The improvement of user interaction through graphical representation of a parallel simulation is the purpose of this study. The requirements for the user's control commands and the effect of these requirements upon the entire system are developed and demonstrated. This research provides an increased capability to assist in battlefield management and critical decision making processes.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Parallel Simulation, Visual Interactive Interface	81
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL